CITCEA

DISEÑO E IMPLEMENTACIÓN DE SOFTWARE APLICADO A COMUNICACIÓN Y CONTROL DE DATOS SOBRE ELECTRÓNICA DE POTENCIA

Trabajo Final de Grado

José Ignacio Bustamante Vargas

Director: Samuel Galceran Arellano
Ponente: Ernest Teniente Lopez
Especialidad: Ingeniería del Software
20 de Marzo del 2019

Índice

1 Introducción

Este documento se presenta como la memoria sobre el trabajo de final de grado que lleva por título “Diseño e implementación de Software aplicado a comunicación y control de datos en electrónica de potencia”. Trabajo que ha sido realizado en la facultad de informática de Barcelona (FIB) de la Universidad Politécnica de Cataluña (UPC) dentro de la especialidad de Ingeniería del Software.

Cómo el propio título indica, el objetivo del trabajo consiste en el desarrollo de un Software cuya finalidad será controlar dispositivos y gestionar datos pertenecientes a un proyecto industrial del área de la electrónica de potencia [1].

1.1 Motivación del proyecto

La realización del trabajo se hará en el CITCEA-UPC [2], centro de transferencia tecnológica, bajo los términos laborales establecidos en la presente normativa de trabajos de final de grado [3] bajo modalidad B. Y en el grado de ingeniería informática de la facultad de informática de Barcelona.

Por un lado, como trabajador del CITCEA, existe la motivación de desarrollar adecuada y exitosamente todo el software vinculado al centro de manera que este tenga un valor productivo. Concretamente en este trabajo tendremos que desarrollar, por demanda de un nuevo cliente del CITCEA, el software vinculado a un nuevo proyecto.

Por el otro lado, como estudiante de la FIB, hay un interés académico que consiste en satisfacer los objetivos propios de un trabajo de final de grado, así como consolidar los conocimientos adquiridos durante el transcurso del grado de ingeniería informática y la especialidad de ingeniería del Software en un entorno práctico y laboral.

Por último, añadir que también existe un interés personal de cara a adquirir experiencia práctica en el desarrollo de proyectos y software mejorando así mi posible desempeño en futuros trabajos y/o proyectos.

1.2 Contexto

Cómo viene siendo costumbre durante los últimos años el Software cada vez tiene más relevancia en los diferentes sectores económicos de nuestra sociedad y el sector industrial, donde pertenece el CITCEA, no es una excepción.

El CITCEA, en su día a día, se encuentra asociado con empresas y proyectos industriales de los cuales algunos tienen como temática principal investigar e innovar en el campo de la electrónica de potencia (área sobre la que tratará el proyecto sobre el que desarrollaremos nuestro software). Dentro del conjunto del proyecto el software puede

desempeñar los roles de comunicación y gestión de datos; control y sincronización de dispositivos; y optimización de recursos en base a modelos matemáticos predictivos.

1.2.1 El proyecto

El proyecto denominado "BusVar", que hará de piloto del software resultante de este trabajo, constará de dos dispositivos principales: Por un lado, un DSP (*Digital signal processor*); y por el otro lado, un computador que irá empotrado en un armario próximo al DSP (al que nos referiremos a partir de ahora como PCE).

El DSP nos permitirá obtener datos y ejercer control sobre los componentes físicos de una red eléctrica que irá asociada a este. Para nosotros este actuará como un dispositivo de bajo nivel del cual nos abstraeremos en su programación interna (es un elemento que utiliza software ajeno al alcance del trabajo). Para poder interactuar y leer los datos que vaya procesando, así como enviarle ordenes de control, nos comunicaremos con él utilizando un cable y protocolo CAN [4] (*Controller Area Network*). Podemos entenderlo como una conexión TCP (*transmission control protocol*) bidireccional que tendrá una API (*Application Programming Interface*) asociada pero más adelante entraremos en detalles de su especificación e implementación.

El PCE interactúa con la API CAN del DSP con la finalidad de emular su estado en un futuro próximo y en base a los resultados obtenidos enviar las consignas de control que la emulación considere más óptima. y, paralelamente, ofrece una API a partir de un servidor que usa el protocolo Modbus TCP/IP[5] (cuya implementación y funcionamiento ya detallaremos más adelante). Esta última será necesaria para que un dispositivo externo pueda controlar la emulación en caso de que necesite intervención externa.

1.2.2 Situación actual

A día de hoy, en el CITCEA, se han desarrollado múltiples proyectos de índole similar al proyecto BusVar. Siguiendo la metodología de trabajo tradicional la implementación del software del PCE consistiría en implementar las APIs Modbus y CAN para cada una de las operaciones especificadas en la fase de diseño. Para ello se reutilizaría, de proyectos anteriores, librerías GPL (*Generic Public License*) [6] que nos proporcionan operaciones de lectura/escritura y modelos matemáticos de emulación. El código realizado en los proyectos previos funcionaba sobre un PCE de una marca de a día de hoy carece de soporte y fabricación. El desarrollo del código de dichos proyectos se hizo haciendo uso de Qt (conjunto de herramientas gráficas multiplataforma) de manera que se hace muy difícil importar una de las funcionalidades por separado ya que el código está hecho y pensado para trabajar exclusivamente con el entorno gráfico de Qt y para las APIs propias de cada proyecto.

1.3 Actores implicados

A continuación, se mencionarán los actores implicados en el proyecto BusVar. Remarcar nuevamente que en el ámbito del trabajo de final de grado trata exclusivamente del sistema software que funcionará sobre el PCE. Pese a ello, este sigue formando parte de los proyectos desarrollados por el CITCEA de manera que los interesados en el correcto desarrollo del Software son también los interesados en el buen desarrollo de los proyectos en electrónica de potencia del CITCEA.

1.3.1 Equipo de desarrollo

El equipo de desarrollo está formado por 3 ingenieros industriales y un estudiante de ingeniería informática. De los 3 industriales uno se encarga del montaje de la red, especificar las APIs y los dispositivos. Otro programa el entorno del DSP. El último se encarga de supervisar el desarrollo del Software del PCE. Por otro lado, el estudiante de ingeniería informática tendrá que diseñar e implementar el sistema software además de instalar y configurar el entorno del PCE.

1.3.2 CITCEA-UPC

Como beneficiario principal del proyecto tenemos al CITCEA. Es quien ha sido contratado para la realización del mismo por lo que su interés en la finalización adecuada del proyecto es directo.

1.3.3 Clientes del CITCEA-UPC

Quienes contratan al CITCEA para el desarrollo de una tecnología lo hacen porque tienen interés en optimizar/desarrollar sus productos y/o servicios. Son quienes invierten en los proyectos y los mayores interesados en su éxito

1.3.4 Clientes de los clientes del CITCEA-UPC

Como se mencionó en un principio el CITCEA-UPC crea tecnología la cual puede ser usada por sus clientes para ofrecer servicios y/o productos a sus propios clientes. Esto depende mucho de cada proyecto en particular, pero pueden darse casos particulares donde los mayores beneficiarios sean estos últimos.

1.3.5 Universidad Politécnica de Cataluña (UPC)

El CITCEA-UPC está adscrito al departamento de ingeniería eléctrica de la UPC. En consecuencia, la UPC se lleva entre un 20% de la facturación de los proyectos industriales del CITCEA (información facilitada por el propio CITCEA). Lo ya

mencionado añadido a la imagen y prestigio que pueden dar proyectos de calidad por parte del CITCEA hacen que la UPC esté interesada e implicada en ello.

2 Estado del arte

En primer lugar, destacar que dado el ámbito académico y de investigación en el que se realiza este trabajo y por filosofía del centro descartaremos cualquier solución que implique Software privativo ajeno. Se trabajará e importará Software Libre o bien se importará software abierto más software propio.

Tras varios años de experiencia laboral en el CITCEA y una extensa búsqueda de soluciones que podrían simplificar el desarrollo de los proyectos se han encontrado dispositivos Hardware que permiten traducir la comunicación de un bus Modbus a un bus CAN por ejemplo.

También existen ya implementadas APIs de comunicaciones que nos permiten abstraernos del hardware y la generación de los paquetes de datos. Estudiaremos las ventajas y contras que pueden ofrecernos cada una de ellas contra hacer nuestra propia implementación.

Por último, a la hora de diseñar e implementar nuestro software estudiaremos patrones de programaciones que ofrecen soluciones a problemas abstractos. Se hará un ejercicio de abstracción de nuestro software y se verá si es práctico importar a nuestro proyecto el diseño de alguno de estos patrones ya implementados.

2.1 Traductores de protocolos

Podemos encontrar en el mercado la posibilidad de adquirir adaptadores que nos permite adaptar de CAN a Modbus o viceversa. Para ello tenemos que conectar un adaptador físico y hacer uso de un software de configuración proporcionado por el fabricante para ligar/indicar que direcciones de uno equivalen con el otro.

Esta solución podría ser válida para nuestro proyecto. Pero la descartaremos por el coste de adquisición de dichos adaptadores ya que deberíamos comprar uno para cada equipo. Por otro lado, estos solo transformas los mensajes que provienen de una interfaz. Es decir que no nos permiten trabajar con servicios o puertos diferenciados como en nuestro caso donde tendríamos una conexión vía CAN y otra vía Ethernet.

Algunos ejemplos de estos los podemos ver en ADFWEB [7] y AnyBus [8]

2.2 Librerías de comunicación

En el caso de la implementación de los protocolos en sí podemos encontrar más material. Dos claros ejemplos de ellos son Libmodbus [9] y SocketCAN [10]. Ambos trabajan con la librería de sockets de BSD (Berkeley Software Distribution) por lo que

también existe la posibilidad de hacer nuestra propia implementación de los protocoles haciendo uso de esta misma.

Dependiendo del entorno de desarrollo existen librerías equivalentes como QtCANBus[11] para Qt y Python-can[12] en Python.

También existen programas que nos permiten enviar y recibir mensajes a través de una interfaz gráfica a nivel usuario, cómo por ejemplo "QModMaster" [13]. Estas nos pueden ser muy útiles para probar nuestro entorno, pero no disponen de una API que nos permite usarlas desde nuestro código ni tampoco nos facilitan el código para estudiar cómo han implementado el protocolo.

2.3 Patrones de Diseño de Programación

Pese a que el Software que estamos trabajando en crear es específico para los proyectos del CITCEA-UPC los problemas a los que nos enfrentamos no dejan de tener un gran número de factores comunes con otros proyectos.

Los patrones de diseño son un compilatorio de soluciones a problemas abstractos y genéricos en los que nos podemos basar, según los requisitos específicos de nuestro sistema, para hacer nuestro diseño específico. Los patrones que pueden ser útiles a la hora de diseñar nuestro Software en particular, que están listados en [14], son:

2.2.1 Adaptador

Convierte una "interface" de una clase en otra que nuestro sistema pueda entender. Nos permite trabajar con clases que a priori serían incompatibles.

En nuestro caso particular adaptaríamos las "interface" de nuestra aplicación de programación, en inglés llamadas "Application Programming Interface" (API), Modbus y CAN para hacerlas compatibles con nuestro sistema.

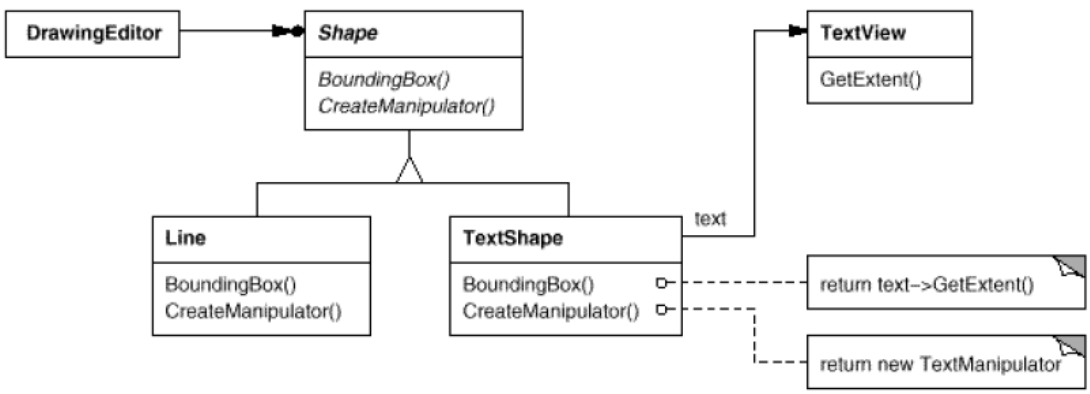


Figura 1 - Modelo de clases del patrón Adaptador extraído de [14].

2.2.2 Cadena de responsabilidades

Permite evadir el acoplamiento entre clases que se envían eventos permitiendo a múltiples objetos gestionar una misma petición. La petición se delega por una cadena de objetos hasta que uno la trate.

Particularmente, deberemos tratar los mensajes que nos lleguen por nuestro bus. Este nos enviará una petición de proceso de mensaje cada vez reciba uno y nos interesa permitir a diferentes objetos de nuestro sistema tener la posibilidad de gestionarlos. También podría ser necesario modificar el diseño original del patrón para no solo delegar el mensaje si no delegar la gestión del mismo en algún caso dado.

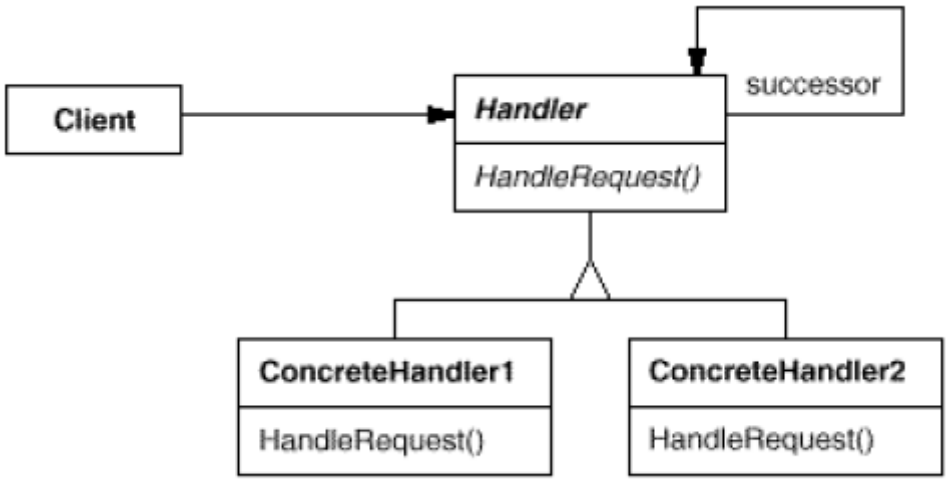


Figura 2 – Modelo de clases del patrón de cadenas de responsabilidades extraído de [14]

2.2.3 Fachada

Nos proporciona una "interface" unificada a un conjunto de "interfaces" a un subsistema. En otras palabras, nos proporciona una "interface" de alto nivel que hace nuestro sistema más fácil de usar.

Concretamente, definiremos aquellas operaciones que los ingenieros del CITCEA-UPC requieren de nuestro sistema de la manera más abstracta posible para que tengan

que saber lo menos posible de cómo está implementado.

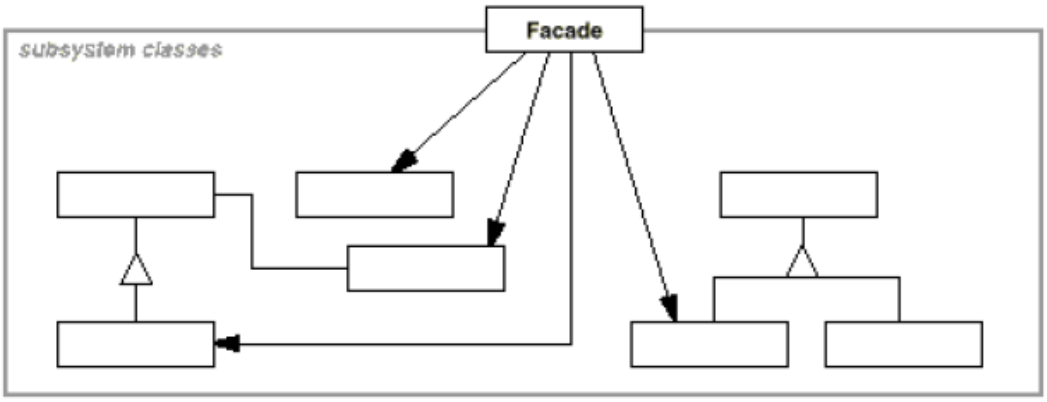


Figura 3 – Modelo de clases del patrón fachada extraído de [14].

2.2.4 Método de fábrica

Define una "interface" para crear objetos. Permite a una clase decidir que instancias de subclases tienen que existir.

Para nuestro Software podremos encontrarnos con la situación, en un futuro, que nuestro sistema operativo no sea Linux o bien que el comportamiento de nuestro bus pueda variar entre cliente o servidor. El patrón de fábrica nos permitirá crear un objeto de bus acorde a las necesidades del proyecto que este en uso o bien crear diferentes buses con el comportamiento necesario en un mismo proyecto.

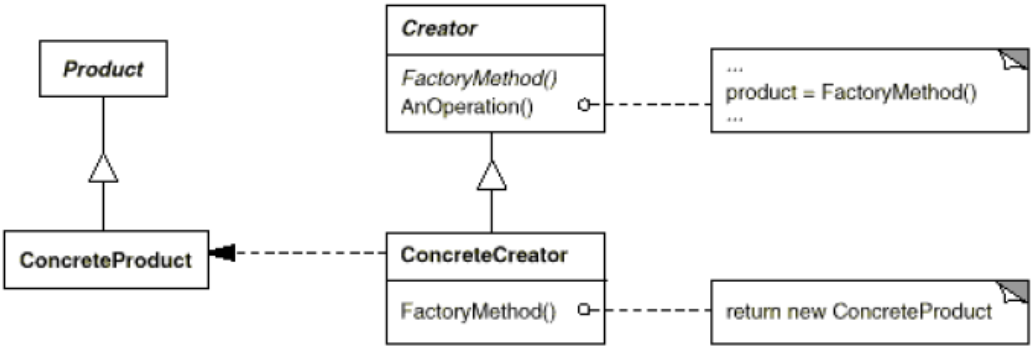


Figura 4 – Modelo de clases del patrón método de fábrica extraído de [14].

2.2.5 Observador

Define una dependencia uno a muchos entre diferentes objetos de manera que cuando el uno cambia de estado todos los demás son notificados automáticamente.

Como existe la necesidad de controlar los dispositivos que están conectados por el bus necesitaremos representar el estado de estos en nuestro sistema. El patrón observador nos permitirá actuar en consecuencia a los cambios que detectemos en el

modelo de los dispositivos de bajo nivel conectados.

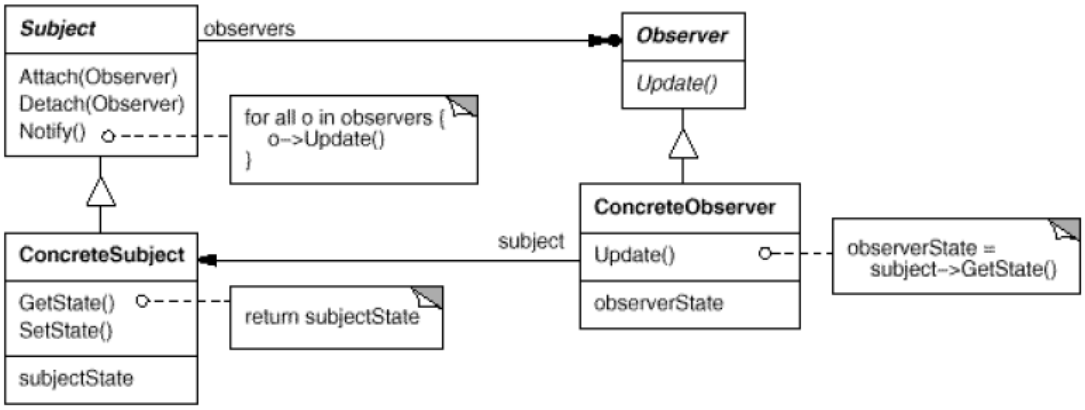


Figura 5 – Modelo de clases del patrón Observador extraído de [14]

2.2.6 Singleton

Asegura que solo pueda existir una instancia de una clase y nos provee de acceso global a esta.

Para acceder a información del sistema suele ser una buena práctica encapsular dichas consultas en una clase "Singleton". También lo usaremos para crear clases controladoras. Como no suele ser una buena práctica tener muchas clases globales encapsularemos todos los "Singleton" dentro de otra clase "Singleton" que se encargue de localizarlos.

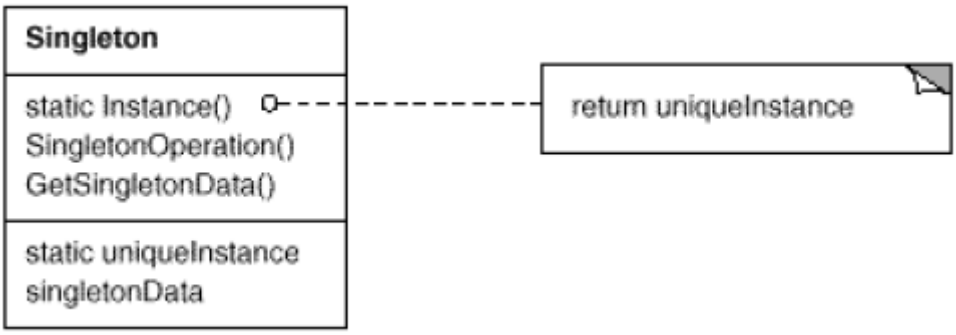


Figura 6 – Modelo de clases del patrón Singleton extraído de [14].

2.2.7 Plantilla

Define la plantilla de un algoritmo en una operación. Esta se divide en subclases las cuales podremos catalogar en dos tipos: Comunes y Sustituibles. En las primeras se implementan aquellas partes del algoritmo de todas las subclases deberán ejecutar de la misma manera. Las segundas están pensada para ser implementada por subclases de forma personalizada.

En nuestro sistema haríamos una plantilla de bus para aquellas partes de código común y las peculiaridades de cada sistema o tipo de bus serían implementadas en su respectiva subclase. También se usaría este patrón para reaprovechar la máxima cantidad de código ya escrito en cualquier jerarquía de clases.

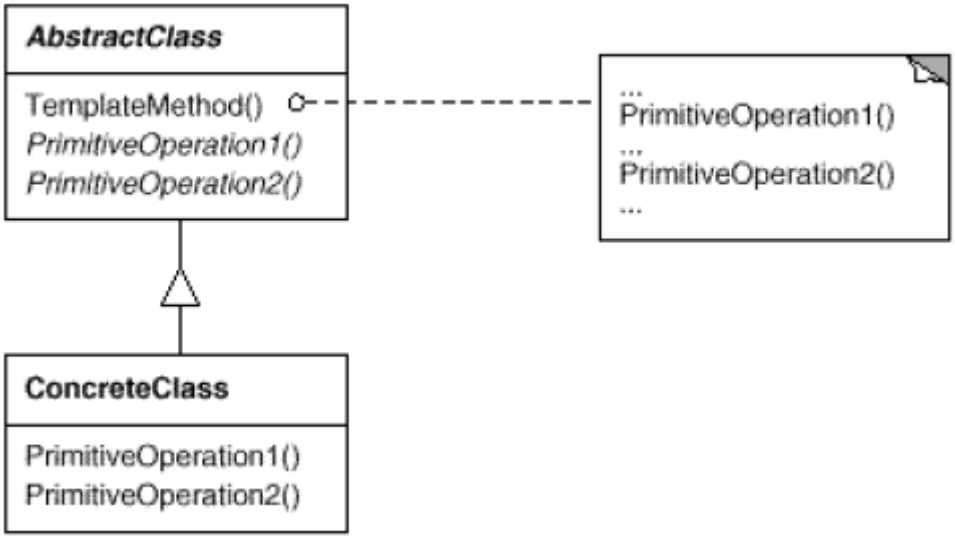
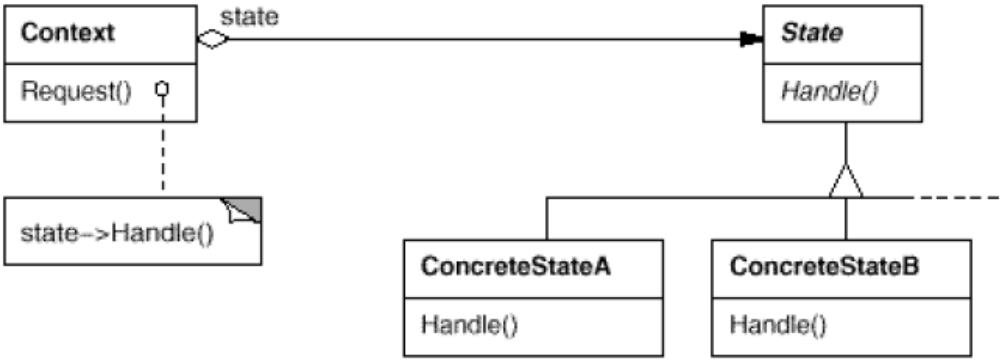


Figura 7 – Modelo de clases del patrón plantilla extraído de [14].

2.2.8 Estado

Permite a un objeto alterar su comportamiento según cambia su estado interno.

Particularmente nos puede ser útil a la hora de gestionar los mensajes entre los diferentes dispositivos ya que nos puede interesar responder de una manera u otra en base a un estado interno del sistema.



3 Formulación del problema

3.1 Problema

De cara al desarrollo del Software podemos identificar los siguientes problemas en la reutilización de código existente en el centro:

Primero, pese a la existencia de numerosos proyectos de carácter similar al BusVar no podemos reutilizar código debido a que el proyecto actual se ha optado por cambiar de fabricante del PCE. Consecuentemente el sistema operativo y el entorno es más potente pero también utiliza versiones más actuales de librerías, entornos de desarrollo y sistema operativo. El trabajo necesario para actualizar el entorno de proyectos anteriores se hace más costoso que empezar uno nuevo.

En segundo lugar, se ha optado por abandonar el uso de Qt y apostar por el uso de GUIs para navegadores WEB dada su mayor portabilidad. El código de proyectos anteriores no solo hacía uso de las librerías gráficas de Qt si no también en la parte lógica y a la hora de usar estructuras de datos (como por ejemplo QVector y Qhttp entre otros).

En tercer y último lugar, para cada una de las operaciones ofrecidas por cada dispositivo se implementaba la respuesta o petición en cuestión. En muchos casos siendo código repetitivo donde solo variaban algunos parámetros y generando cientos de líneas de código condicional y redundante.

De cara al uso de librerías GPL nos podemos encontrar en un escenario donde sería necesario modificar o corregir alguna funcionalidad. Esto nos obligaría a publicar y distribuir dichas modificaciones además de la necesidad de crear una capa de abstracción exclusiva para diferenciar nuestro código del código GPL si no queremos hacer lo mismo con este también.

3.2 Objetivo

El objetivo es desarrollar un Software para el proyecto BusVar que pueda ser ejecutado en un entorno PCE y en un PC (*personal computer*). Además, nuestra meta será priorizar que este sea modular y tenga poco acoplamiento, siempre que sea posible, con la finalidad de reutilizar la mayor cantidad de funcionalidades posibles de cara a futuros proyectos. Este deberá ser capaz de tratar los mensajes de dispositivos externos que se comuniquen con él ya sea vía CAN o Modbus y también de enviarles mensajes. Sumado a esto último tendrá que gestionar los datos que le sean comunicados para una posible monitorización y/o registro.

Otro aspecto importante de nuestro Software será que solo debamos especificar una vez la API sobre la que trabajaremos y que el mismo software se encargue de adaptar ésta a las APIs de los dispositivos externos.

3.3 Alcance

El proyecto consiste en el diseño e implementación de un Software para el CITCEA-UPC. Este se diseñará de forma que puede incrementar y añadir nuevas funcionalidades. Por lo que haremos una primera iteración de 2 meses (marzo y abril de 2019) y luego una segunda iteración de un mes (mayo de 2019). La primera requiere de mucho más tiempo debido a que habrá que preparar el entorno de trabajo, así como la instalación y configuración del sistema sobre el que operará nuestro software.

Este deberá funcionar en un entorno PCE o en un computador industrial que tendrá conexión local con los dispositivos de bajo nivel. También tendrá conexión a una red privada donde podría ofrecer un servicio a aplicaciones externas sean o no desarrolladas por el CITCEA sin que debamos priorizar aspectos de ciberseguridad como encriptar los mensajes. Requerirá de un fichero de configuración o una estructura de dato de diccionario como entrada en la cual estará especificada los mensajes que se pueden realizar a los dispositivos de bajo nivel. El lenguaje de programación elegido para ello será C++ ya que nos permite usar la orientación objetos a la vez que es compatible con el lenguaje C que es como están escritas una gran parte de las librerías que importaremos a nuestro sistema. La evolución de las arquitecturas ARM nos permiten disponer de más recursos en entornos de PCEs y descartar una implementación pura en C enfocada puramente en el rendimiento de la ejecución y la gestión de memoria del programa. Pero aun así sigue siendo necesario disponer de ciertos puntos donde podamos tratar la memoria como veamos convenientes y C++ consigue muy bien un equilibrio entre un desarrollo modular y orientado a objetos (aunque popularmente no sea el lenguaje favorito para ello dada su sintaxis y requerimientos de conocimientos avanzados de programación para sacar un provecho real de estos puntos).

En otras palabras, nuestro sistema se limita a intermediar con los dispositivos de bajo nivel y elementos externos (por ejemplo, un ingeniero haciendo uso de una interfaz gráfica o un cliente a una hipotética API proporcionada por nuestro Software). Estos últimos podrían consultar los datos y estados de los dispositivos asó como solicitar la ejecución de comandos de configuración o acción de estos y/o el propio PCE.

Para conseguirlo se mantendrá una conexión TCP bidireccional y asíncrona entre el PCE y el DSP. Al mismo tiempo deberemos implementar un servidor Modbus/TCP que será el encargado de responder las peticiones de dispositivos/clientes externos. El Software del PCE se encargará de controlar, sincronizar y gestionar las peticiones y comunicaciones de ambos lados.

3.4 Requisitos Funcionales

- **Monitorizar el DSP:**
  Cada segundo el DSP enviará por el bus CAN datos sobre su estado que el PCE tendrá que usar para decidir que operaciones se pueden realizar o no.
- **Establecer conexión con el servidor Modbus:**
  Se podrá establecer una conexión con un servidor Modbus/TCP
- **Cerrar conexión con el servidor Modbus**
  El cliente Modbus se podrá desconectar cuando lo vea conveniente
- **Enviar parámetros de configuración**
  El cliente Modbus podrá escribir datos en el servidor Modbus que serán delegados al DSP para indicarle el escenario sobre el que deberá actuar.
- **Enviar orden de ejecución**
  El cliente Modbus podrá enviar ordenes que se le transmitirán al DSP para que este pueda conocer cuándo se han enviado todos los parámetros de configuración, reiniciar la simulación, empezar a enviar datos, etc…
- **Consultar parámetros de configuración**
  El cliente Modbus puede consultar los parámetros de configuración que están escritos en el instante de la petición.
- **Consultar magnitudes de los sensores**
  El cliente Modbus puede consultar los datos leídos a tiempo real por los sensores del DSP.

3.5 Requisitos No Funcionales

En este punto enumeraremos aquellos requisitos no funcionales en base a la especificación de Volere que tendrá que satisfacer nuestro sistema:

- ***Ease of Use*** **[Volere 11a]:**
  El software deberá ser fácil de usar para aquellos usuarios que estén familiarizados con el uso de APIs Modbus.
- ***Learning*** **[Volere 11c]:**
  Será necesario que el usuario aprenda o tenga acceso a la especificación de las operaciones ofrecidas.
- ***Accessibility*** **[Volere 11e]:**
  El Software es perfectamente utilizable por personas daltónicas y/o discapacidad auditiva.
- ***Speed and latency*** **[Volere 12a]:**
  El Software deberá ser capaz de leer y escribir todos los elementos de las tablas Modbus y sus equivalentes en CAN cada segundo.
- ***Reliability and Availability*** **[Volere 12d]:**

El Sistema deberá ser capaz de mantenerse en funcionamiento las 24 horas del día los 365 días del año.

- ***Robustness or Fault-Tolerance* [Volere 12e]:**
  El Software deberá ser tolerante a pérdidas de conexión y capaz de reconectarse cuando está vuelva.
- ***Robustness or Fault-Tolerance* [Volere 12e]:**
  El Sistema deberá arrancar el Software cuando arranque y recuperar el estado en que se encontraba en el momento del apagón.
- ***Capacity* [Volere 12f]:**
  El servidor Modbus solo tendrá las conexiones de un PC de control y/o un usuario que quiera configurar el DSP.
- ***Longevity* [Volere 12h]:**
  El sistema deberá operar y ser mantenido como máximo durante 15 años.
- ***Interfacing with Adjacent Systems* [Volere 13c]:**
  El Software deberá interactuar con un DSP vía CAN y con un cliente Modbus/TCP.
- ***Adaptability* [Volere 14c]:**
  Se ha previsto que el software deba ser capaz de funcionar para arquitecturas de procesadores arm64, armhf y amd64.
- ***Access* [Volere 15a]:**
  El sistema solo será accesible desde una red privada por un usuario técnico.

3.6 Metodología y rigor

La metodología de trabajo a seguir estará mayormente inspirada en la conocida "Extreme Programming" [16]. Para ello se harán iteraciones como en la imagen que podemos observar a continuación:

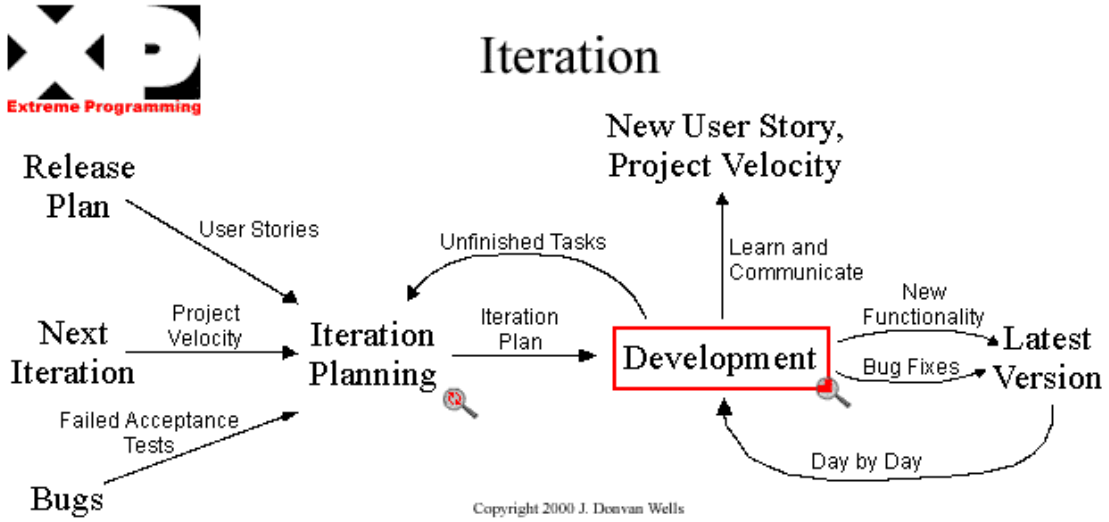


Figura 8 – Representación de una iteración en "Extreme Programming" [16]

Por cada iteración se analizarán los requisitos de la funcionalidad a implementar. Acorde al análisis se realizará un primer diseño especificado en "Unified Model Language" (UML) [17]. Este se hará únicamente del modelo de clases de nuestro sistema. Además, se especificará un trasfondo de los requisitos de nuestro sistema tal como se aconseja en [18] inspirado en la metodología de "SCRUMBAN". A partir de este se empezarán a implementar las clases con código descriptivo y comentado. El modelo y el código siempre pueden verse "refactorizados", que se define en [19] como la edición de código sin cambiar la funcionalidad mejorando aspectos de diseño, durante el desarrollo de otra clase o funcionalidad si aportan una nueva funcionalidad o mejora a la hora de implementarlas. Por cada clase y funcionalidad implementada se hará una validación de esta y todas las anteriores para asegurarnos de que el sistema funciona como esperamos.

Para la primera iteración antes de la implantación del código tenemos que considerar el tiempo de preparación del entorno de trabajo (Instalación y configuración de herramientas para el desarrollo, monitorización y validación del Software; los entornos virtuales sobre los que se desarrollará el Sistema y el entorno de producción donde se ejecutarán las versiones estables).

Es necesario destacar que si alguna funcionalidad requiere del uso de herramientas externas o ha de ser importada porque existe una alternativa ya implementada por terceros se hará la respectiva instalación y configuración sobre los entornos de desarrollo y una vez validados sobre el de producción en la iteración correspondiente.

Por último, se hace un proceso de escucha que consiste en hablar con los clientes para obtener una orientación si el camino que se está siguiendo es el adecuado y el producto satisface sus necesidades o bien, si existe algún cambio a realizar debido a que el resultado no era del todo como este lo esperaba. Tras la valoración de la iteración se procedería a especificar los requisitos de la siguiente repitiendo el ciclo.

3.6.1 Herramientas y entorno de desarrollo

Ahora haremos un listado y una breve descripción de las herramientas que usaremos durante el desarrollo del proyecto (todas serán de código abierto):

- Eclipse CDT

  Integrated development Environment (IDE) con licencia abierta que nos permite desarrollar aplicaciones en C++ facilitándonos la faena de compilar, crear entornos de pruebas y generar ejecutables.
- GOOGLE Test

  Librería de código abierto desarrollada por google para hacer juegos de prueba unitarios y mocks.
- VirtualBox.

  Es una aplicación que nos permite administrar máquinas virtuales. Lo usaremos para crear los entornos de trabajo reusables y aislados durante el desarrollo del proyecto.
- Git.

  Es Software de control de versiones que nos facilitará la vida a la hora de recuperar el estado de nuestro Software en versiones previas y o bien integrar nuevas funcionalidades [20].
- DEBIAN

  Distribución de Linux y sistema operativo que estará instalado en el "embedded".
- GNU g++.

  Compilador de C++ para sistemas Linux-GNU que es el lenguaje que usaremos para el proyecto piloto del Software. Nuestro software se deberá poder compilar con las versiones para arquitecturas ARM y AMD64.

- Libmodbus.

  Librería que utilizaremos a la hora de implementar el adaptador de Modbus a nuestra API y viceversa.
- SocketCAN

  Librería que deberemos usar para hacer uso de la interfaz de CAN en Linux.

3.6.2 Herramientas para monitorizar la evolución

Para monitorizar y hacer el seguimiento del proyecto usaremos un servidor privado de GitLab. Este es una aplicación WEB que viene con Git integrado. Nos proporciona una interfaz WEB que nos provee de seguimiento de errores, una wiki para documentar, estadísticas sobre la actividad y evolución de nuestro software y herramientas de CI/CD (Continuous Integration and Continuous Delivery) [21]. En resumen, un entorno web con todas las herramientas que necesitaremos para monitorizar nuestro Software y validarlo de forma continuada asegurándonos que siempre sea funcional en el entorno de producción.

3.7 Método de Validación

Para cada funcionalidad y clase de nuestro Software se harán unos juegos de prueba unitarios. Para ello especificaremos cuales son las salidas y resultados que nuestro sistema debería responder dada unas entradas específicas. Estos se ejecutarán cada vez que se adhiera contenido nuevo asegurándonos de que este funciona como esperamos al mismo tiempo que nos aseguramos también que todos los anteriores siguen funcionando igual.

Pasado los juegos de pruebas, generada la documentación y probada la ejecución del Software en un entorno de producción se dará por válida una historia de usuario (terminología definida en [19] para referirnos a una iteración en particular).

3.8 Obstáculos y riesgos

Como en todo proyecto Software siempre existen inconvenientes que en el peor de los casos podrían acabar con un proyecto.

3.8.1 Bugs

La existencia de “BUGs” o, en otras palabras, de comportamiento no esperado en la ejecución de nuestro Software. Para simplificar la existencia de estos usamos los métodos de validación (pero dada nuestra condición humana de cometer errores) y la naturaleza compleja de los sistemas informáticos es inevitable que encontremos alguno. En particular la complejidad de su resolución aumenta cuando estos los encontramos en librerías que se han importado.

La solución consiste en tener un protocolo de acción a la hora de detectarlos. Este consistirá en valorar la prioridad y gravedad. Dependiendo de este factor se decidirá en que versión del Software deberá corregirse para que esta pueda validarse. En el peor de los casos sería necesario crear una rama paralela de la versión actual en producción en GIT y corregir el error allí antes que se incorpore cualquier nueva funcionalidad y subirla lo antes posible al entorno de producción.

3.8.2 Tecnología

En nuestro proyecto tenemos dos elementos tecnológicos limitantes que debemos tener en cuenta.

En primer lugar, las especificaciones del hardware donde se ejecutará nuestro Software. Por ejemplo, a la hora de negociar los requisitos de nuestro sistema se puede acordar con el cliente que deberá ser capaz de ver el estado de los dispositivos de bajo nivel cada 10 milisegundos. Para poder afirmar si este requisito se puede añadir tenemos

que tener consciencia si la tecnología usada para el proyecto en concreto es suficiente y en caso de serlo el software deberá ser capaz de sacarle el rendimiento apropiado.

En segundo lugar, los medios de transmisión de datos están limitados por su infraestructura. Basándonos en el ejemplo anterior, si los requisitos son suficientes pero la latencia y ancho de banda de nuestro cable CAN conectado a un dispositivo de bajo nivel no, aunque el software potencialmente pueda asumirlos el sistema no sería capaz.

La solución pasa por especificar y comprar el hardware necesario para cumplir las expectativas del cliente o bien no acordar requerimientos que el hardware que se usará no puede asumir.

3.8.3 Calendario

Como en todo proyecto siempre estamos sujetos a que las estimaciones sobre el coste temporal de una tarea no sean acertadas. Esto puede derivar en que las horas necesarias sean más de las previstas aumentando el coste real del proyecto.

La solución es tener un presupuesto de margen para cubrir estos gastos extras además de asesorarse con personas con experiencia en proyectos de índole similar.

3.8.4 Diseño y especificación

Una mala especificación y diseño del software implica que este no estaría solucionando el problema para el que ha sido creado o al menos no lo estaría haciendo eficientemente. Este encarecería innecesariamente el proyecto o en el peor de los casos lo volvería inviable.

La solución es tener acuerdos escritos, claros y precisos sobre los requisitos del sistema y contar con un ingeniero del Software que se encargue de hacer el diseño.

4 Entorno de desarrollo y producción

A continuación, se detallará la configuración e instalación tanto del entorno de desarrollo y de producción.

El desarrollo ha sido principalmente en uno de los despachos del CITCEA en un PC con Windows 10 Professional. Este estará conectado por un cable serie al PCE y el desarrollo del Software se hará en una máquina virtual usando un DEBIAN 9 gracias al software de Oracle VirtualBox.

4.1 Entorno de la máquina virtual

Aquí trataremos la configuración e instalación de la máquina virtual. Para ello se instalará usando la ISO amd64 de la página oficial de DEBIAN 9 [22] (también conocida como DEBIAN Stretch).

En este entorno la instalación se hará con una única partición conservando la mayoría de valores por defecto ya que el sistema propiamente no se deberá mantener.

Una vez instalado el sistema se procederá a la instalación de los paquetes necesarios para poder desarrollar y compilar el Software. Se usará el gestor de paquetes de DEBIAN apt. Dicho sistema operativo de caracteriza por un centro de paquetes de software estable. Es decir que muchas veces tardan en integrar las últimas novedades y versiones ya que los desarrolladores priorizan la estabilidad del sistema y por ese mismo motivo instalaremos las versiones que ya vienen por defecto de los siguientes programas:

- **Gnu gcc compiler:** para poder compilar para la máquina virtual.
  # apt-get install build-essential
- **Gnu gdd debugger**: para poder debuggar el código haciendo uso del menú de herramientas del IDE.
  # apt-get install gdd
- **Git:** para realizar el control de versiones del software, la importación/exportación del código y disponer de backups del código
  # apt-get install git
- **Eclipse CDT:** para desarrollar, debugar, compilar y ejecutar los juegos de prueba del Software. Aparte de instalarlo se creará un acceso directo en el path /usr/local/bin para poder ejecutarlo desde cualquier directorio sin tener que escribir todo el path.
  # descargar el IDE desde la web oficial [23].
  # tar –zxvf elFicheroDescargado.tar.gz
  # sudo mv elFicheroDescargado /opt
  # sudo ln –s /opt/elFicheroDescargado/eclipse /usr/local/bin/eclipseCPP4.11.0
  # eclipseCPP4.11.0

- **Googletest:** para hacer los juegos de pruebas unitarios y mocks en nuestro IDE.
  # apt-get install googletest
  # apt-get install google-mock
- **Arm Linux gnueabihf gcc:** para compilar los binarios que deberá ejecutar el PCE.
  # sudo dpkg –add-architecture armhf
  # sudo apt-get update
  # sudo apt-get install curl build-essential crossbuild-essential-armhf -y

En el directorio de trabajo de eclipse se crearán dos proyectos. El primer proyecto, denominado BusVar, será el que se configure con la finalidad de crear el ejecutable que irá al entorno de producción. Conjuntamente habrá otro proyecto llamado BusVarTest que utilizará el framework de googletest junto a plugin del ECLIPSE proporcionado por los propios desarrolladores de googletest. Este último deberá incluir los binarios generados por la compilación del proyecto BusVar por lo que su compilación y ejecución es dependiente del primer proyecto. El motivo de esta separación es debido a que eclipse hace uso de autotools de GNU [24] y este no permite generar un makefile con dos main. El framework de googletest requiere una función main que le invoque adecuadamente y para no tener que cambiar constantemente entre uno u otro se ha decidido separarlos en dos proyectos.

4.2 Entorno del PCE

En el PCE que se usará para el proyecto BusVar se hará uso de un sistema operativo DEBIAN Stretch para armhf que descargaremos de la wiki de los propios fabricantes que nos lo ha proveído [25]. Este contiene preinstalado gcc, vim, X11, slim y un autologin con un entorno gráfico xfce4.

La instalación se hará en una tarjeta SD que irá incorporada al PCE posteriormente. Para conseguirlo insertaremos la SD el PC de desarrollo y le indicaremos a la máquina virtual que capture el driver. Una vez insertada la SD en nuestra máquina virtual el sistema operativo le asignará una unidad de disco automáticamente. Los sistemas Linux etiqueta los dispositivos de memoria en el directorio /dev/ según se van añadiendo con los nombres de sda, sdb, sdc, sdd, etc. Suponiendo que al conectar nuestra SD le asigna la unidad sdc los pasos a seguir son:

- asegurarnos con la herramienta fdisk que contiene una única partición de memoria en formato ext3.
- Crear el sistema de ficheros en la partición única previamente creada.
  VM# mkfs.ext3 /dev/sdc1.
- Montar la unidad en un directorio nuevo por ejemplo /mnt/sd/.
  VM# sudo mount /dev/sdc1 mnt/sd
- Descomprimir el sistema proveído por los fabricantes.
  VM# sudo tar –numeric-owner –xjf debían-armhf-stretch-latest.tar.bz2 –C /mnt/sd

- Desmontar y sincronizar los buffers para asegurar que toda la memoria se guarda correctamente.
  VM# sudo umount /mnt/sd
  VM# sync

En este punto solo se deberá insertar la tarjeta SD en el PCE y este arrancará el sistema que le acabamos de cargar.

Una vez arrancado el sistema se creará un usuario nuevo para evitar trabajar como root en todo momento y poder conectarnos remotamente al PCE a partir de este para evitar comprometer al sistema en caso de una incidencia de seguridad o error humano.

Además de los programar preinstalados se añadirán un servidor Secure Shell (SSH) y un servidor Network File System (NFS).

El primero nos permitirá conectarnos con un terminal remoto en caso en caso de que exista la necesidad de acceder ya sea para mantener, arreglar o actualizar el sistema sin tener que ir físicamente donde se encuentre el PCE. La instalación y configuración realizadas han consistido en:

- PCE# apt-get install openssh-server
- cambiar el puerto 22, que es el usado por defecto según podemos comprobar en *The Internet Assigned Numbers Authority* (IANA) [26], por uno superior al 1024.
- Limitar el acceso exclusivamente a un usuario sin privilegios en el sistema ya que es relativamente fácil acceder a las credenciales de conexión con un monitor de red. Este escalaría de credenciales según lo fuese necesitando haciendo uso del comando “su”.

El segundo nos permitirá montar vía red el sistema de ficheros del PCE en nuestro sistema de desarrollo por ejemplo en /mnt/PCE/. Nos será útil a la hora de compilar para el PCE ya que deberemos configurar los directorios de compilación del IDE para que utilice las librerías de gcc ubicadas en el PCE y no las propias de la máquina virtual. También nos puede permitir generar el ejecutable directamente en un directorio en el sistema de ficheros del PCE para poder ejecutarlo desde este. La instalación y configuración realizadas:

- PCE# apt-get install nfs-kernel-server portmap
- Editar el fichero /etc/exports para permitir exclusivamente la conexión de cliente en el rango privado de la subred del despacho de todo el sistema ficheros desde /root añadiendo la siguiente línea:
  / 10.X.X.0/24(rw,no_root_squash,subtree_check)
  Nota: por privacidad en vez de indicar el identificador de red privado usamos el carácter X.
- Cuando de finalice el desarrollo o se configure el PCE que se enviará con el cliente este servicio se deberá eliminar del sistema ya que no aportará ningún tipo de valor añadido.

El PCE consta de una pantalla táctil que nos permite interactuar con el sistema. En caso de ser necesario la integración de una interfaz de usuario para el Software o para la configuración de los parámetros de red del PCE utilizaríamos una funcionalidad ya integrada y preinstalada conocida como modo kiosco. Para ello instalaremos un servidor web ligero lighttpd. Para arrancar el sistema en dicho modo para que el usuario no tenga acceso al sistema propiamente deberemos ejecutar el siguiente script cuando se inicie:

```
#!/bin/bash
chromium-browser –window-size=1024,600 –kiosk localhost –user-data-dir ~/ &
exec xfce4-terminal
```

El sistema viene pre configurado por el fabricante para ejcutar el script automáticamente al iniciar el PCE si lo ubicamos en el directorio /etc/alternatives/ con los permisos de ejecución y de acceso adecuados.

El servidor como la interfaz web consumarán la mayor parte de los recursos del sistema y se deberá monitorizar adecuadamente el impacto que tendrán sobre el Software. Aun así, en base a la experiencia de proyectos previos, haciendo uso de otro PCE con peores especificaciones y más antiguo respondía y cumplía adecuadamente aun con la carga del servidor web por lo que no se espera que sea un factor limitante durante el desarrollo del proyecto.

5 Especificación e implementación

En este punto se verá la especificación y diseño de cada una de las versiones del Software, así como una descripción de la implementación del mismo con algunos fragmentos de código que no comprometan las políticas de privacidad.

Tras una primera reunión con el equipo de desarrollo del proyecto y los clientes se decidió que las funcionalidades más importantes a implementar en primer lugar serían el servidor Modbus y la comunicación de datos por CAN.

La idea original consistía en usar librerías ya implementadas tanto para Modbus como CAN. El problema surgió en el momento de integrar libmodbus que acoplaba fuertemente la gestión de conexiones con el tratamiento de los mensajes dificultando el proceso de usar nuestro propio servidor e integrar exclusivamente aquellas funciones encargas de la codificación y decodificación de los mensajes Modbus. Esto sumado al hecho de que hay varios reportes de bugs en lo que respecta a Modbus TCP y que el proyecto lleva más de un año sin actualizarse se ha optado por hacer una implementación propia en base a la especificación oficial de Modbus.org.

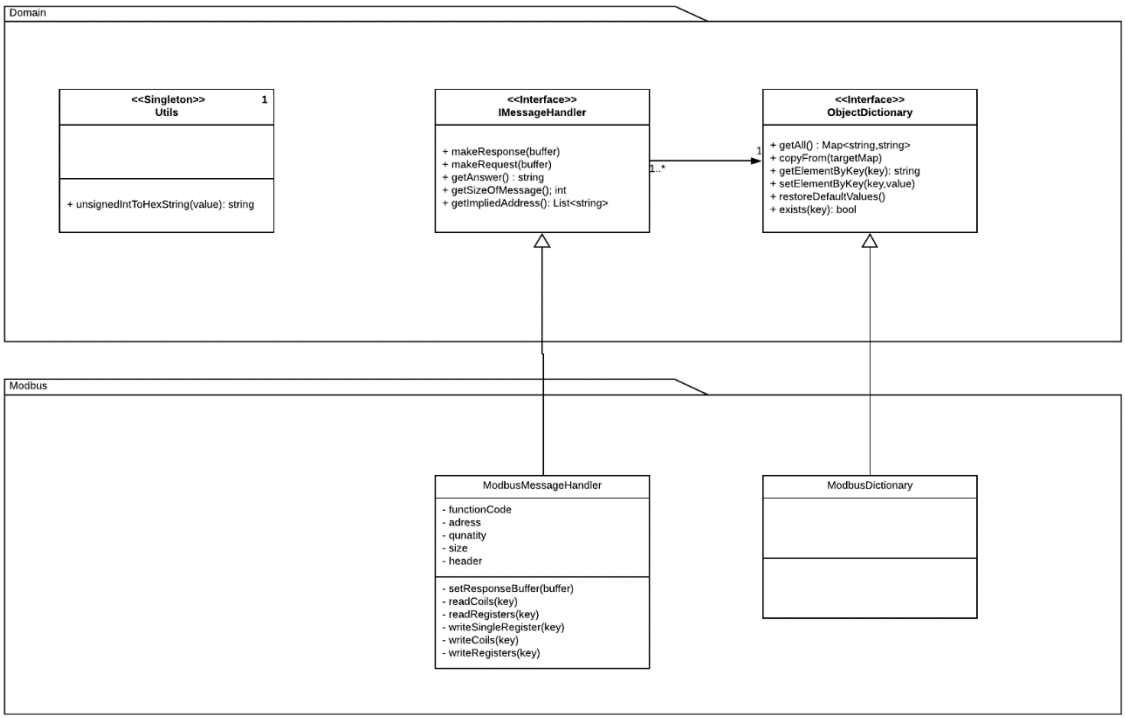
El criterio de versiones del Software está inspirado en la política de versiones usada en DEBIAN. El primer digito nos indicaría que el software tiene suficiente entidad para ser considerado un programa diferente e independiente entre sus diferentes versiones. El segundo digito es indicativo de la adición de una funcionalidad nueva. Y el tercer digito se usaría para las actualizaciones que solo incluyen correcciones de errores o bugs críticos.

5.1 Versión 0.1.0. Gestor de mensajes

En esta primera versión se implementará una interfaz genérica para la codificación y decodificación de mensajes. Una clase concreta que la implemente para el protocolo Modbus. Esta irá asociada a una clase Diccionario de objetos que contendrá los datos que recibamos o enviemos por los mensajes con sus respectivos filtros en caso de ser necesarios.

Como la implementación del código usará una gran variedad de conversión de elementos a tipos de datos no convencionales será necesario la incorporación de librerías útiles que nos faciliten el operar con estos. Encapsularemos el acceso a esta en una clase singletone.

En la imagen mostrada a continuación podemos observar el modelo de clases.



La implementación del gestor de mensajes consiste en consultar o actualizar los campos del diccionario de objetos dependiendo del contenido de la petición del cliente. Este contendrá un código de función que será indicativo de cuál de los métodos deberemos invocar. Cada uno de los datos que podemos encontrar en la petición Modbus estará codificado por bytes en “Little Endian” pero el orden entre cada byte será en “Big Endian”.

La especificación de que debe responder cada mensaje la podemos encontrar en Modbus Application Protocol V1 1b3 6.

La validación y juegos de pruebas unitarios se han inspirado en el propio documento del protocolo Modbus que incluye ejemplos de peticiones y el valor que se esperaría como respuesta.

En las imágenes de abajo podemos ver el código de uno de los juegos de y el código que atiende la petición de la función Modbus “read coils”.

```
  virtual void SetUp()
  {
      MBAP[0] = 0x15;
      MBAP[1] = 0x01;
      MBAP[2] = 0x00;
      MBAP[3] = 0x00;
      MBAP[4] = 0x06;
      MBAP[5] = 0x00;
      MBAP[6] = 0xFF;

      dict = new BusVar::ModbusDictionary("");
  }

  virtual void TearDown() {
  }

};


  TEST_F(ModbusMessageTest,readCoils)
  {
      unsigned char request[7+5];
      std::memcpy(request,MBAP,7);
      request[7] = 0x01;
      request[8] = 0x00;
      request[9] = 0x13;
      request[10] = 0x00;
      request[11] = 0x13;
      ModbusMessage msg(dict); // @suppress("Abstract class cannot be instantiated")
      msg.makeResponse(request); // @suppress("Method cannot be resolved")
      unsigned char expectedResult[7+5] = {0x15,0x01,0x00,0x00,0x06,0x00,0xFF,0x01,0x03,0xCD,0x6B,0x05};
      unsigned char* realResult = msg.getRawMessage();
      ASSERT_TRUE(not std::memcmp(realResult,expectedResult,7+5));
  }
```

```
void ModbusMessage::readCoils(std::string key)
{
    responseBuffer[8] = (unsigned char)std::ceil(this->quantity / 8.0);
    if(not responseBuffer[8]) ++responseBuffer[8];
    ++size;
    values = std::vector<unsigned short>(this->quantity,0);
    unsigned char mask = 0;
    for(unsigned short addr = this->address; addr < this->address+this->quantity; ++addr)
    {
        unsigned short aux = std::atoi(this->dictionary->getElementByKey(key+"<"+std::to_string(addr+1)+">").c_str());
        values[(unsigned char)(mask/8)] |= aux << (mask%8);
        ++mask;
    }

    for(unsigned short i=0; i< (unsigned short)responseBuffer[8];++i)
    {
        responseBuffer[i+9] = values[i];
        ++size;
    }
}
```

El código de arriba a partir de la lectura de un buffer que vendrá de un socket, aunque para los juegos de pruebas introducimos manualmente ejemplos de posibles valores que nos llegarían. El gestor de mensajes decodificaría la cabecera en base al protocolo Modbus la cual nos indicará el tamaño y número de datos a solicitar, así como la operación que el cliente desea realizar. El método “makeResponse” se encargaría tratar posibles errores de formato o incumplimiento del protocolo. En caso de no existir errores se invocaría el método asociado al código de función que el cliente solicita. El método de tratamiento de función del ejemplo accede a los valores almacenados en el diccionario y los codifica en un buffer de respuesta para cumplir con el estándar Modbus.

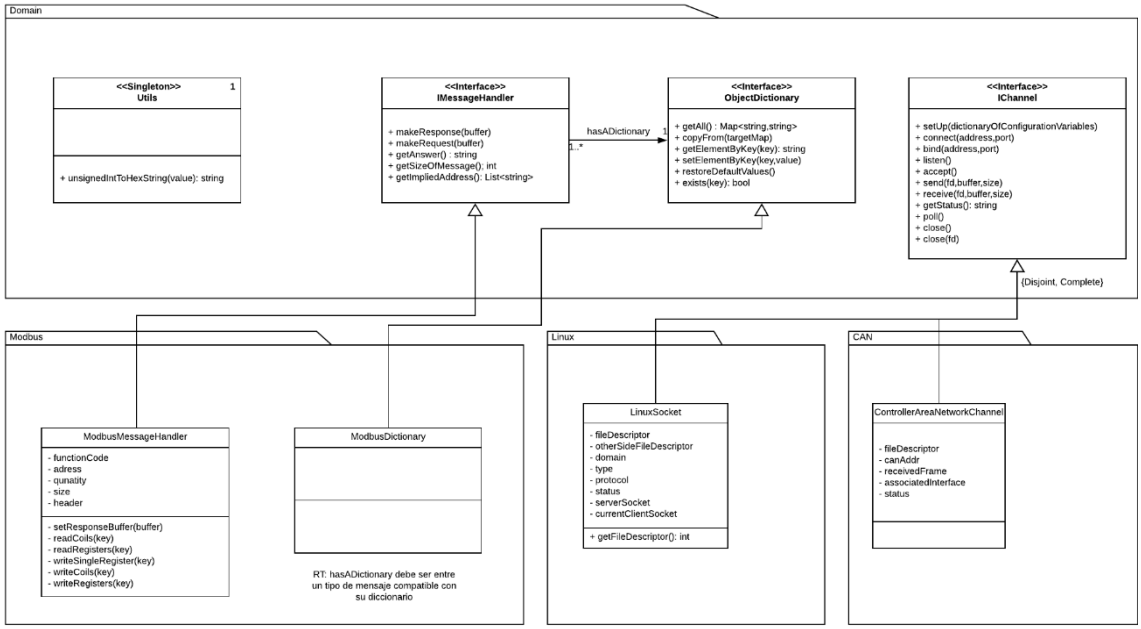
El diccionario Modbus usado para los juegos de pruebas esta inicializado en su constructora con los valores que estos esperan leer.

5.2 Versión 0.2.0. Canal de comunicación

Una vez disponemos de la funcionalidad para tratar los mensajes vía protocolo Modbus el siguiente paso a seguir es establecer los canales de comunicaciones. Para ello se usará una interfaz genérica de canal de comunicación la cual tendrá unas cabeceras basadas en la API de Berkeley Sockets. La idea es abstraer las funcionalidades lógicas y la gestión de las comunicaciones del entorno en el que nos encontramos. Es decir que si el cliente o algún futuro proyecto requieren el uso de canales en un sistema operativo diferente de Linux o por canales alternativos de comunicación similares a can simplemente se tenga que re implementar en forma de nuevas clases concretas la interfaz.

Para el proyecto BusVar implementaremos dos medios. En primer lugar, usando la propia librería de sockets de Linux inspirada también en la Berkeley. Por último, haciendo uso de una librería incluida en el kernel de Linux llamada Socket CAN

El diagrama de clases actualizado es el siguiente:



La implementación de las clases concretas consiste en invocar y configurar adecuadamente las librerías usadas. A continuación, veremos algunas imágenes con ejemplos del código del proyecto. En necesario remarcar que estas clases pretenden ser un medio de acceso a los canales, pero no gestionarlos. Es decir que el tratamiento y la gestión de posibles errores deberá ser tratada por un elemento controlador que invoque la interfaz. Pese a ello, con la finalidad de conseguir que el Software funcione en el menor tiempo posible, las llamadas se harán con los parámetros de configuración que serán necesarios para el proyecto propiamente. En caso de que exista la necesidad de cambiar alguno de refactorizaría y parametrizaría.

```
BusVar::LinuxSocket::LinuxSocket(int domain, int type, int protocol)
{
    // TODO Auto-generated constructor stub
    fileDescriptor = ::socket(domain,type,protocol);
    std::cerr << "fd -> " << fileDescriptor << std::endl;
    this->domain = domain;
    this->protocol = protocol;
    this->type = type;
    this->sizeOfClientsQueue = 1;
    this->clientFileDescriptor = -1;
    if(fileDescriptor == -1)
        status = strerror(errno);
    else
        status = "CONFIGURED";
}

void BusVar::LinuxSocket::setUpServer(char* address,int port)
{
        //Copy the string withAddress to a C string format in address and setting up the structs
        bzero((char*)&serverSocket,sizeof(serverSocket));
        serverSocket.sin_addr.s_addr = inet_addr(address);
        serverSocket.sin_family = domain;
        serverSocket.sin_port = htons(port);
        //.....................................................................
}

int BusVar::LinuxSocket::bind(const std::string withAddress,int port)
{
    //Init variables

    //.................................................

    //Copy the string withAddress to a C string format in address and setting up the structs
    setUpServer(const_cast<char*>(withAddress.c_str()),port);
    //.....................................................................

    return ::bind(fileDescriptor, (struct sockaddr *)&serverSocket, sizeof(serverSocket));
}


/**
 * config the current socket as passive (ready for accept incoming connections)
 */
int BusVar::LinuxSocket::listen()
{
    return ::listen(fileDescriptor,sizeOfClientsQueue);
}

/**
 * accept incoming connection
 */
int BusVar::LinuxSocket::accept()
{
    std::cerr << "fd accept -> " << fileDescriptor << std::endl;
    socklen_t lenght = sizeof(currentClientSocket);
    clientFileDescriptor = ::accept(fileDescriptor, (struct sockaddr*) &currentClientSocket,&lenght);
    return clientFileDescriptor;
}

/**
 * send data to the socket
 */
int BusVar::LinuxSocket::send(int fileDescriptor,unsigned char* data, int size)
{
    const char* aux = reinterpret_cast<const char*>(data);
    return ::send(fileDescriptor,aux,size,0);
}

/**
 * read data from the socket
 */
int BusVar::LinuxSocket::receive(int fileDescriptor,unsigned char* data, int size)
{
    return ::recv(fileDescriptor,data,size,MSG_WAITFORONE);
}
```

```
ControllerAreaNetworkChannel::ControllerAreaNetworkChannel() {
    // TODO Auto-generated constructor stub
    status = "OK";
    fileDescriptor = ::socket(PF_CAN, SOCK_RAW, CAN_RAW);
    if(fileDescriptor < 0)
    {
        this->close();
        status = "socket error";
    }
}

ControllerAreaNetworkChannel::~ControllerAreaNetworkChannel() {
    // TODO Auto-generated destructor stub
    this->close();
}

void ControllerAreaNetworkChannel::setUp(std::string interface, int bitrate)
{
    std::string command = "ip link set "+interface+" down";
    ::system(command.c_str());
    command = "ip link set "+interface+" type can bitrate "+std::to_string(bitrate);
    ::system(command.c_str());
    command = "ip link set "+interface+" up";
    ::system(command.c_str());
}


int ControllerAreaNetworkChannel::bind(const std::string withAddress, int port)
{
    std::strcpy(associatedInterface.ifr_name,withAddress.c_str());
    ioctl(fileDescriptor,SIOCGIFINDEX,&associatedInterface);

    canAddr.can_family = AF_CAN;
    canAddr.can_ifindex = associatedInterface.ifr_ifindex;

    int err = ::bind(fileDescriptor, (struct sockaddr*) & canAddr, sizeof(canAddr));
    if( err < 0)
        status = "binding error";
    return err;
}

int ControllerAreaNetworkChannel::listen()
{
    return 0;
}

/**
 *
 */
int ControllerAreaNetworkChannel::accept()
{
    return 0;
}

/**
 * send data to the socket
 * fd is ignored at this implementation
 */
int ControllerAreaNetworkChannel::send(int fd,unsigned char* data, int size)
{
    int err = ::send(fileDescriptor,data,size,0);
    return err;
}

/**
 * read data from the socket
 */
int ControllerAreaNetworkChannel::receive(int fd,unsigned char* package, int size)
{
    int err = ::recv(fileDescriptor,package,sizeof(struct can_frame),MSG_WAITFORONE);
    return err;
}
```

Por la naturaleza de las conexiones las pruebas realizadas han hecho uso de software externo. En el caso de CAN se han usado las propias herramientas incluidas en Linux de Socket CAN cansend y cansniffer. Estas nos permiten enviar mensajes y monitorizar tanto los mensajes recibidos como los enviados. En el caso de las conexiones tcp de Linux se ha usado un cliente Modbus de Matlab [27] ya que a efectos prácticos el protocolo Modbus TCP es una conexión TCP.

Los juegos de pruebas solo se limitarían a comprobar que se han recibido los datos y enviado además de establecido la conexión sin devolver ningún tipo de error.

```
TEST_F(ModbusTCPTest,acceptConnection)
{
    int fd = sock->listen(); // @suppress("Method cannot be resolved")
    sock->accept(); // @suppress("Method cannot be resolved")
    EXPECT_TRUE(fd > -1);
}

TEST_F(ModbusTCPTest,isConnected)
{
    sock->connect(); // @suppress("Method cannot be resolved")
    std::cout << sock->getStatusOfSocket() << std::endl; // @suppress("Method cannot be resolved") // @suppress("Invalid overload")
    EXPECT_TRUE(sock->getStatusOfSocket() == "CONNECTED"); // @suppress("Method cannot be resolved")
}
```

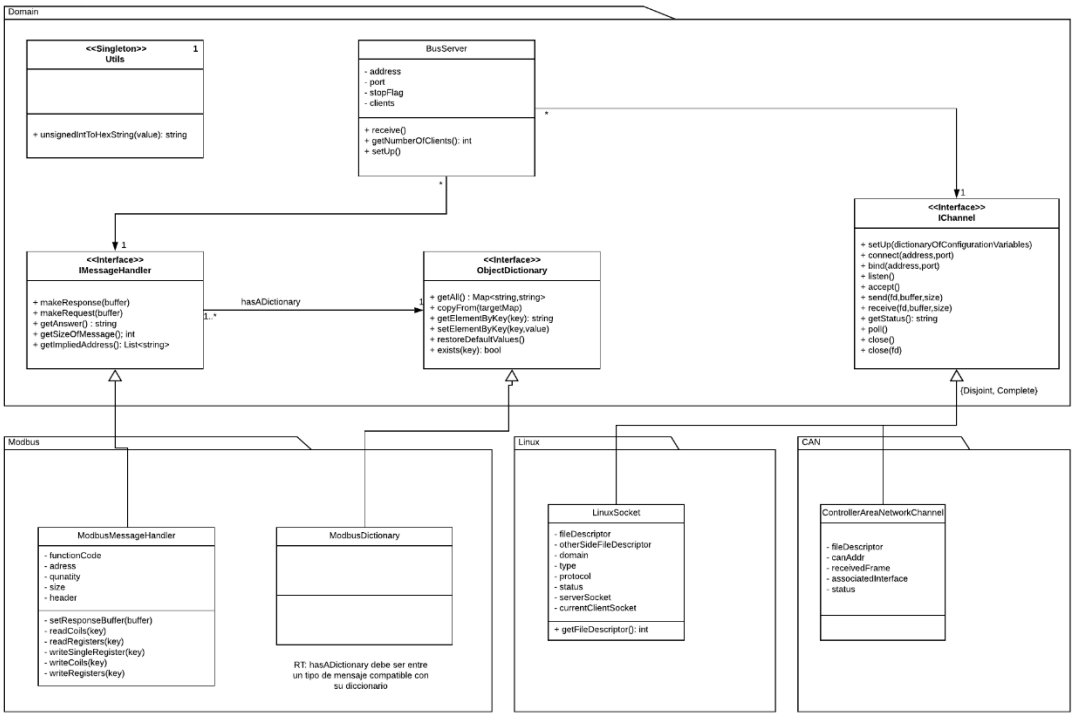
5.3 Versión 0.3.0. El Servidor

En esta versión solo se considerará atender una conexión con un único cliente simultáneamente ya que en los requisitos actuales del proyecto no se especifica lo contrario. Sin embargo, se tiene previsto a la hora de estructurar la clase que en un futuro debamos incluir una cola de conexiones.

Otro aspecto importante a considerar en esta versión es la inclusión de concurrencia en nuestro Software ya que el servidor deberá escuchar y tratar las peticiones en un hilo de ejecución separado. El PCE donde se ejecutará el Software está limitado a un core por lo que la ejecución será secuencial. Aun así, como no se garantizará el orden de ejecución de los hilos separados se usarán semáforos para asegurar la escritura y lectura de datos en los diccionarios en forma de transacciones.

El servidor se limitará a responder mensajes en base a un gestor de mensajes por un canal asociado. Este se encuentra totalmente abstraído del protocolo usado o el medio físico.

El diagrama de clases y la implementación las podemos encontrar en las imágenes a continuación:




```
BusServer::BusServer(configMap* config,IChannel* concretChannel, IMessage<std::string,unsigned short int>* concretMessage) {
    // TODO Auto-generated constructor stub
    clients = 0;
    channel = concretChannel;
    configuration = config;
    messageType = concretMessage;

    address = getMapValueByKey("SERVER_ADDRESS");
    port = std::atoi(getMapValueByKey("SERVER_PORT").c_str());

    stopFlag = false;

    stateMachine = NULL;
    channel->bind("0.0.0.0",port);
    serverTask = new std::thread(&BusServer::runServer,this);
}

void BusServer::runServer()
{
    channel->listen();
    int fd = channel->accept();
            if(fd > -1) {
                ++clients;
            }
    while(not stopFlag)
    {
        this->receive(fd);
    }
        channel->close(fd);
}


std::string BusServer::getMapValueByKey(std::string key)
{
    configMap::const_iterator it = configuration->find(key);
    if(it == configuration->end()) {
        return "";
    }
    else {
        return it->second;
    }
}

void BusServer::receive(int fd)
{
    std::cout << " receiving" << std::endl;
    std::vector<unsigned char> buffer(300);
    int err = channel->receive(fd,&buffer[0],300);
    if(err < 0) {
        return;
    }
    messageType->makeResponse(&buffer[0]);
    err = channel->send(fd,messageType->getAnswer(),messageType->getSizeOfMessage());
    if(err < 0) {
            return;
        }
```

5.4 Versión 0.4.0. El diccionario CAN

El diccionario CAN se caracteriza por tener elementos de 8 bytes en Little Endian. Sin embargo, en la especificación de este podemos encontrar sub elementos que varían de tamaño de manera irregular. Para poder acceder y gestionar el acceso se hará uso de una Union. Esta nos permitirá acceder a una misma zona de memoria en base a estructuras que la segmentan de forma diferente. Esta manera de estructurar el diccionario será más simple y legible que usar máscaras y desplazamientos de bits.

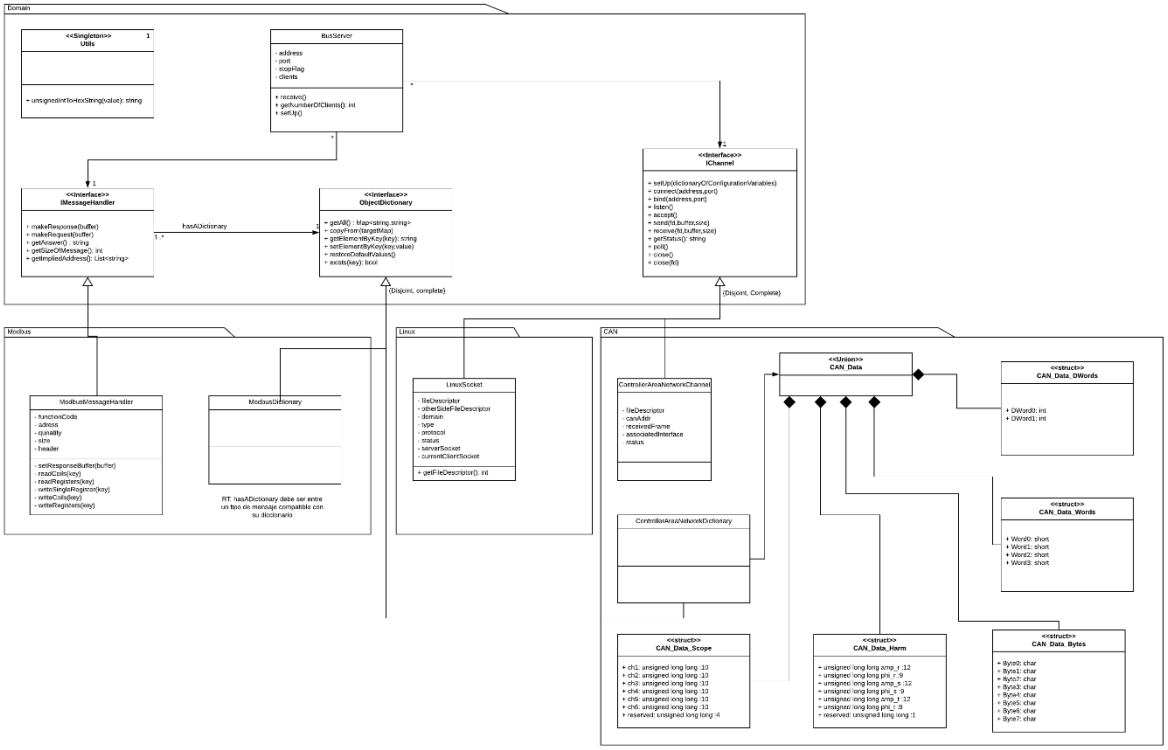
```
union CAN_Data
{
    struct CAN_Data_Bytes   Bytes;  // 8 bit words
    struct CAN_Data_Words   Words;  // 16 bit words
    struct CAN_data_DWords  DWords; // 32 bit words
    struct CAN_Data_Harm    Harm;   // Harmonic setpoint data map
    struct CAN_Data_Scope   Scope;  // Scope data map
    unsigned long long      all;    // 64 bit
};
```

```
struct CAN_Data_Harm {
    unsigned long long amp_r            :12;
    unsigned long long phi_r            :9;
    unsigned long long amp_s            :12;
    unsigned long long phi_s            :9;
    unsigned long long amp_t            :12;
    unsigned long long phi_t            :9;
    unsigned long long reserved         :1;
};

struct CAN_Data_Scope {
    unsigned long long ch1              :10;
    unsigned long long ch2              :10;
    unsigned long long ch3              :10;
    unsigned long long ch4              :10;
    unsigned long long ch5              :10;
    unsigned long long ch6              :10;
    unsigned long long reserved         :4;
};
```

Las comunicaciones por CAN no se basan en el paradigma cliente-servidor para este proyecto en particular. Sin embargo, una vez implementado el diccionario CAN crear un servidor que atienda peticiones en base al protocolo sería simplemente implementar el gestor de mensajes e instanciar las clases concretas pertinentes.

El diagrama de clases sería el siguiente:

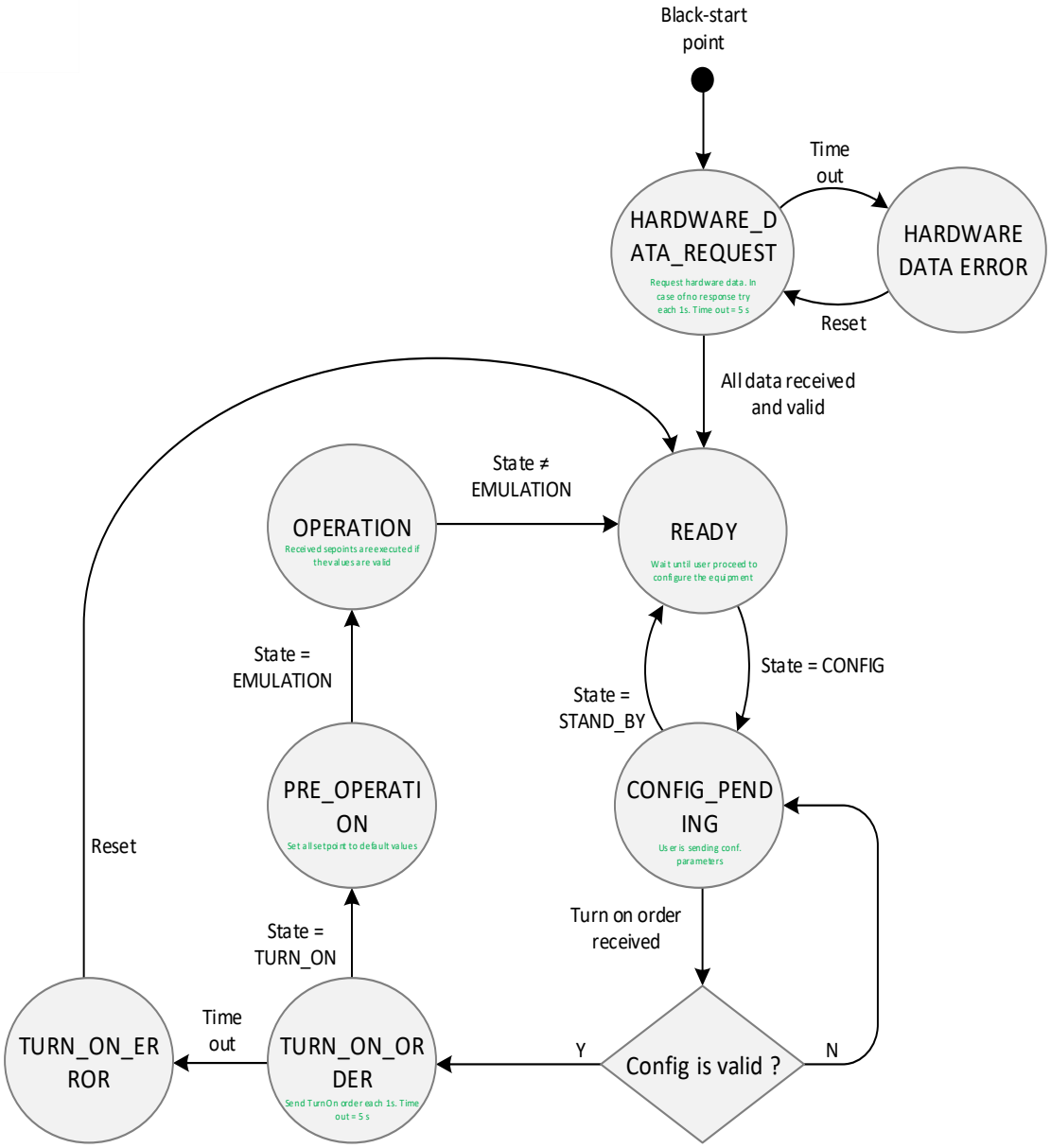


5.5 Versión 0.5.0. La Máquina de estados

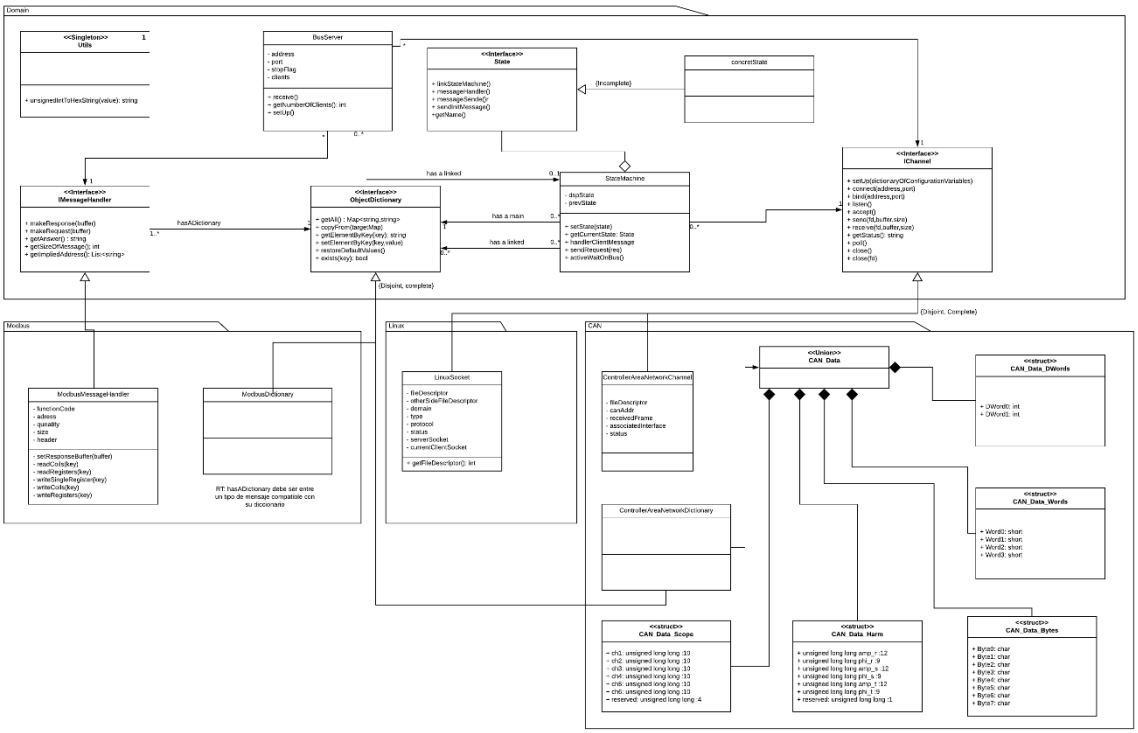
La comunicación entre el PCE y el DSP para el proyecto BusVar utiliza el protocolo CAN y establece un canal asíncrono y bidireccional. Se ha especificado que el PCE deberá interactuar con el DSP en base a un estado interno por lo que nos basaremos en el patrón estado expuesto 2.2.8.

En esta versión se añadirá la estructura del patrón expuesto pero la implementación de los diferentes estados se dejará para la siguiente.

Los estados para el proyecto específico los podemos ver representado en el siguiente grafo, así como sus posibles cambios:



El diagrama de clases resultante sería el siguiente:



Cada estado tendrá tres métodos a implementar de forma opcional que se invocarán cuando sucedan diferentes tipos de eventos.

El primer evento sería cada vez que leemos un mensaje por el bus CAN se distinguirá si es un mensaje de monitorización y ser tratado de forma genérica en la máquina (ya que el tratamiento será independiente del estado interno). En caso de que el mensaje no este catalogado como de monitorización se le delegaría al estado en cuestión. Al igual que en el caso del servidor los eventos de la máquina de estado se ejecutan en hilos concurrentes.

```
void StateMachine::activeWaitOnBus()
{
    while(true)
    {
        if(currentState) {
            std::vector<unsigned char> buffer(sizeof(struct can_frame),0);
            associatedChannel->receive(0,&buffer[0],sizeof(struct can_frame));
            int auxState = monitorCAN((can_frame*)&buffer[0]);
            if(auxState != -1) currentState->messageHandler(buffer,mainDictionary,linkedDictionary,dspState);
        }
    }
}
```

A continuación, podemos ver cómo se gestiona el mensaje que contiene el estado interno del DSP con el que nos estamos comunicando. Existe un total de 12 mensajes de monitorización cuya gestión es similar al caso de la imagen de abajo, pero por confidencialidad no se mostrarán en este documento.

```
int StateMachine::monitorCAN(can_frame* frame)
{
    lock->lock();
    ControllerAreaNetworkDictionary::CAN_Data canLine;
    std::memcpy(&canLine.all,frame->data,frame->can_dlc);
    int state = -1;
    switch(frame->can_id)
    {
        case 256:
            unsigned short bgbv, timerLSB, timerHSB, errorOcurrence1;
            std::memcpy(&bgbv,frame->data,2);
            std::memcpy(&timerLSB,&frame->data[2],2);
            std::memcpy(&timerHSB,&frame->data[4],2);
            std::memcpy(&errorOcurrence1,&frame->data[6],2);

            mainDictionary->setElementByKey("0x100",std::to_string(canLine.all));
            linkedDictionary->setElementByKey("InputRegister<49>",std::to_string(bgbv));
            linkedDictionary->setElementByKey("InputRegister<50>",std::to_string(timerLSB));
            linkedDictionary->setElementByKey("InputRegister<51>",std::to_string(timerHSB));
            linkedDictionary->setElementByKey("InputRegister<52>",std::to_string(errorOcurrence1));

            state = bgbv & 0xFF00;
            state = state >> 8;

            if(prevState != state) {
                std::cout << "state -> " << state << std::endl;
                dspState = state;
            }
            prevState = state;
            break;
```

El segundo evento es el instante en que el servidor actualiza su diccionario asociado por su gestor de mensajes. Este implica un proceso de refactorización en el servidor donde deberemos ligarlo a una máquina de estado para poder asociar dicho evento. Esta asociación se implementará de manera que no sea obligatoria mantenerla, pero para este proyecto deberá existir siempre.

```
void BusServer::receive(int fd)
{
    std::cout << " receiving" << std::endl;
    std::vector<unsigned char> buffer(300);
    int err = channel->receive(fd,&buffer[0],300);
    if(err < 0) {
        return;
    }
    messageType->makeResponse(&buffer[0]);
    err = channel->send(fd,messageType->getAnswer(),messageType->getSizeOfMessage());
    if(err < 0) {
            return;
        }
    if(stateMachine)
    {
        std::list<std::string> address = messageType->getImpliedAddress();
        while(not address.empty()) {
            std::string addr = address.front();
            address.pop_front();
            stateMachine->handlerClientMessage(addr);
        }
    }
}
```

En caso de recibir el mensaje Modbus del tipo “Holding register” cuya dirección este identificada por el valor “1” el tratamiento será independiente del estado.

```
void StateMachine::handlerClientMessage(std::string id)
{
    std::cout << "handling " << id << std::endl;

    ControllerAreaNetworkDictionary::CAN_Data canData;
    struct can_frame frame;
    if(id.find("<1>")!=std::string::npos) {
        canData.all = std::stoull(mainDictionary->getElementByKey("0x11"));
        std::cout << linkedDictionary->getElementByKey(id) << std::endl;
        canData.Words.Word0 = (unsigned short) std::stoul(linkedDictionary->getElementByKey(id));
        mainDictionary->setElementByKey("0x11",std::to_string(canData.all));
        std::memcpy(frame.data,&canData.Words.Word0,2);
        frame.can_id = 0x11;
        frame.can_dlc = 2;
        currentState->messageSender(associatedChannel,mainDictionary,linkedDictionary,id);
        associatedChannel->send(0,(unsigned char*)&frame,sizeof(struct can_frame));
        linkedDictionary->setElementByKey("HoldingRegister<1>",std::to_string(0));
    }
    else currentState->messageSender(associatedChannel,mainDictionary,linkedDictionary,id);
}
```

Por último, nos encontramos con la situación de tener que enviar o tratar mensajes en el instante en que cambiamos el estado interno del PCE.

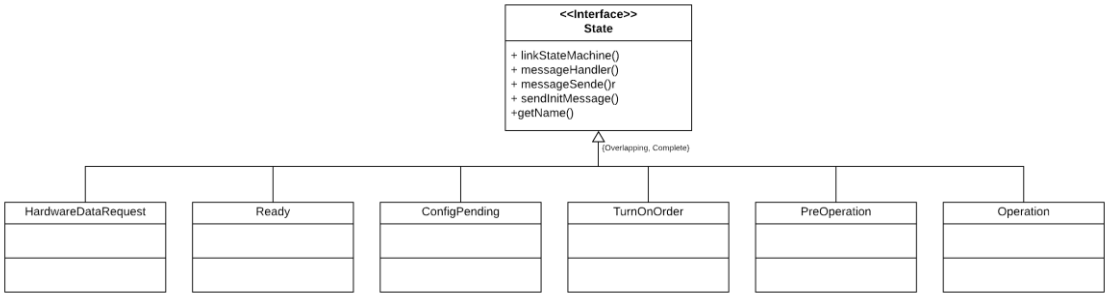
```
void StateMachine::setState(State* newState)
{
    if(currentState) delete currentState;
    currentState = newState;
    currentState->locker = lock;
    currentState->linkStateMachine(this);
    taskInit = new std::thread(&State::sendInitMessage,currentState,associatedChannel,mainDictionary,linkedDictionary);
}

State* StateMachine::getCurrentState()
{
    return currentState;
}
```

5.6 Versión 0.6.0. Adición de los estados

Esta versión se centra en la implementación de los estados internos que puede adquirir la máquina de estados según la especificación del grafo de estados que se ha enseñado en el apartado anterior.

El diagrama de clases se mantiene similar por lo que solo mostraremos como cambia la herencia de la interfaz de estados.



Cada uno de los estados nos permite tratar o ignorar los diferentes mensajes que vamos recibiendo tanto por Modbus TCP o CAN.

Como la especificación del mapa CAN será parte del convertidor eléctrico que el CITCEA venderá a su cliente por motivos de confidencialidad no se detallarán las direcciones y los mensajes enviados en detalle.

En este punto solo se informará el tipo de tratamiento realizado en cada estado:

- HardwareDataRequest:
  Captura el evento de mensajes CAN para consultar los parámetros con la información física del DSP (su código de serie, valores de potencia, frecuencia, etc) y el inicio de estado para comunicarle al DSP que se ha iniciado el programa y en consecuencia solicitar los valores que deberemos capturar.
- Ready:
  Solo captura los mensajes del Bus CAN. Este método tiene como funcionalidad esperar que el DSP se sincronice adecuadamente. Cuando detecta que el DSP se encuentra en un estado adecuado para recibir parámetros de configuración cambia de estado.
- ConfigPending:
  Capturamos los mensajes del Bus CAN para detectar si el DSP ha cambiado de estado de manera que debamos sincronizarnos con este ya sea para volver al estado previo o pasar al siguiente. También captura un mensaje modbus que nos indica que el servidor ya contiene todos los parámetros de configuración listo que será indicativo de enviar un mensaje al DSP para proseguir al siguiente estado.
- TurnOnOrder:
  Cuando entramos en este estado deberemos enviar todos los mensajes de configuración almacenados en el servidor Modbus por el Bus CAN en el formato correspondiente. También captura los mensajes del bus can para sincronizar un posible cambio de estado según el estado del DSP.
- PreOperation:
  Cuando entramos en este estado reinicializamos los elementos de la tabla Modbus especificados de manera local. Este estado también monitoriza un posible cambio de estado en el DSP para saber cuándo deberá cambiar de estado.
- Operation:
  Como todo los estado anteriores monitoriza el estado del DSP para sincronizar un posible cambio de estado. También puede recibir la orden por parte del DSP de reinicializar un conjunto de valores del diccionario Modbus. En este estado capturamos un rango de mensajes recibidos por el servidor y se delegan al DSP haciendo la codificación correspondiente.

5.7 Versión 1.0.0. BusVar

Esta versión añade los estados de errores a los que entraríamos si el DSP no responde alguno de los valores que esperamos. Una vez dentro de este estado simplemente esperaríamos a recibir un mensaje de reinicio por parte del DSP que nos indicaría cuando vuelve a encontrarse en un estado operativo.

También se añade la capacidad reconectarse si existiera algún problema de conexión. El comportamiento del programa en tal caso sería pasar a un estado de error e intentar reconectarse indefinidamente hasta conseguirlo.

6 Calendario inicial

6.1 Estimación de la duración del proyecto

La estimación de la duración del proyecto es de 4 meses y 2 semanas. Concretamente desde el mes de Febrero (18/02/2019) hasta finales de Junio (20/06/2019).

6.2 Consideraciones

El proyecto puede verse aplazado por la existencia de días festivos, vacaciones del personal y posibles huelgas.

Se ha decidido por incluir únicamente dos iteraciones por el contexto de proyecto de final de grado. El diseño del Software y la metodología de trabajo en todo momento está enfocada a añadir nuevas funcionalidades e iteraciones, y a realizar mantenimiento de las funcionalidades ya existentes.

En caso de errores o una mala previsión el proyecto finalizará en las fechas establecidas. El impacto se verá en las funcionalidades que tenga la versión resultante al acabar el proyecto (pueden verse aumentadas o disminuidas).

7 Descripción de las tareas

Nuestro proyecto Software está pensado y estructurado para formar parte de otros proyectos de Electrónica de Potencia. Este hecho implica que nuestro Software debe ser fácilmente escalable, moldeable y configurable dados unos requisitos específicos de cada proyecto en particular.

Las necesidades y requisitos de cada proyecto, dada la naturaleza de los clientes, puede variar en cualquier momento dado y en consecuencia tenemos que estar preparados para adaptarnos a dichos cambios. Para conseguir tal propósito la metodología de desarrollo del proyecto estará basada en “Extreme Programming”.

Toda metodología inspirada en “Agile” tiene como característica principal el desarrollo de un Software funcional e incremental a partir de iteraciones. Estás duran un ciclo basado en puntos relativos (la equivalencia de punto a horas puede ajustarse para ser más próxima a la realidad del proyecto). Cada iteración viene a representar una funcionalidad nueva y posibles reajustes a las ya existentes para la adecuada integración de esta última.

Se denomina “Sprint” a un conjunto de iteraciones que se realiza durante un periodo de tiempo previamente especificado que en su conjunto nos dan un software con un numero de funcionalidades suficientes para considerar que tenemos una nueva versión.

7.1 Tareas

Podemos dividir el proyecto en 5 grandes bloques los cuales veremos a continuación:

7.1.1 Trabajo inicial

En primer lugar, es necesario definir el contexto y alcance del proyecto, trazar un plan y estudiar su viabilidad. Para ello requeriremos de 3 semanas de trabajo aproximadamente y se las siguientes tareas:

1. Contexto y alcance del proyecto.
2. Planificación del proyecto.
3. Presupuesto y sostenibilidad.

Los recursos utilizados serán:

- Recursos Humanos
  - Gerente de proyecto
- Recursos “Harware”
  - Ordenador laboral (Intel Core i7-8700 3.2GHz, 16GB RAM, 128GB SSD, 1TB HDD)
- Recursos “Software
  - Windows 10 y Microsoft Word 2016
  - TeamGantt

7.1.2 Configuración y Entorno

En segundo lugar, es necesario analizar los requisitos del sistema, adecuar nuestro entorno de trabajo y preparar el entorno de producción para el correcto desarrollo del proyecto. La duración prevista será de poco más de 3 semanas y se dividirá en 4 tareas:

1. Análisis de requisitos del Sistema.
2. Diseño UML del modelo.
3. Instalación y configuración del entorno de trabajo.
4. Instalación y configuración del entorno de producción.

Los recursos utilizados serán:

- Recursos Humanos
  - Administrador de Sistemas
  - Ingeniero del Software
  - Gerente de proyecto
- Recursos “Hardware”

- Ordenador laboral (Intel Core i7-8700 3.2GHz, 16GB RAM, 128GB SSD, 1TB HDD)
- "PC EMBEDDED" TS-TPC-7990 [28]

- Recursos "Software"
  - Lucidchart
  - Trello
  - Oracle VirtualBox
  - DEBIAN

7.1.3 Gestor del Bus (Sprint 1)

En este bloque ya empezaremos a trabajar en la producción de código. Se han definido 8 tareas y se suma al entorno de trabajo una aplicación para el uso de control de versiones con GIT (GitLab [29]). Las tareas para este primer Sprint serán aquellas cuyas funcionalidades estén asociadas con el Bus de comunicación con un dispositivo de bajo nivel.

Los recursos utilizados serás:

- Recursos Humanos
  - Ingeniero del Software
  - Administrador de Sistemas
  - Gerente de proyecto
  - Programador
  - "Tester"
- Recursos Hardware
  - Ordenador laboral (Intel Core i7-8700 3.2GHz, 16GB RAM, 128GB SSD, 1TB HDD)
  - "PC EMBEDDED" TS-TPC-7990
  - DiskStation DS1817+
- Recursos Software
  - GIT
  - GitLab
  - DEBIAN
  - Eclipse CDT
  - Lucidchart
  - Trello
  - Oracle VirtualBox

11.1.4 Diccionario de Objetos (Sprint 2)

A continuación, se procederá al Sprint 2 donde se implementarán todas aquellas funcionalidades que estén relacionadas la capa lógica del diccionario de objetos. Para conseguirlo se necesitarán exactamente los mismos recursos que en el Sprint 1.

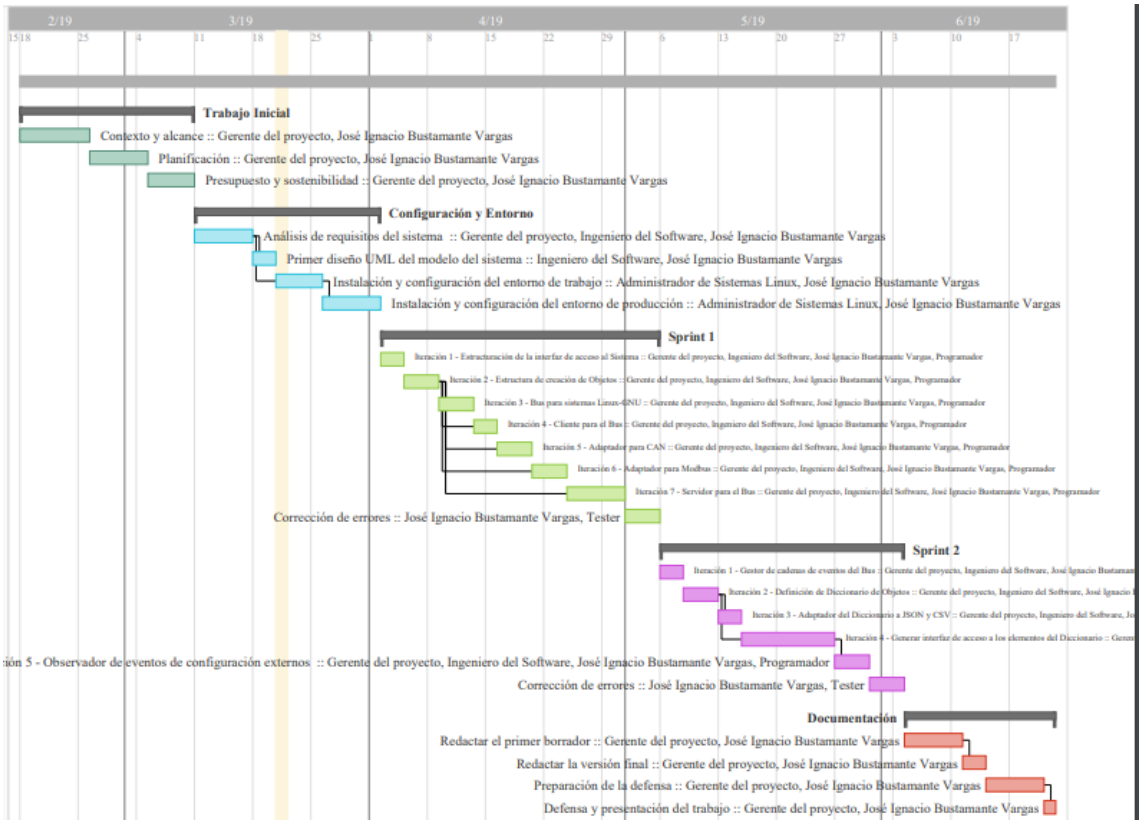
7.1.5 Documentación

Por último, será necesario presentar la documentación completa del proyecto y realizar una defensa de cómo ha ido delante de un tribunal y alguno de los clientes implicados. Las tareas necesarias para conseguirlo serán la redacción de la memoria y la preparación de una presentación para la defensa.

Los recursos utilizados serán los mismos que el bloque del trabajo inicial añadiendo los siguientes Software:

- Lucidchart
- Microsoft PowerPoint

7.2 Diagrama de Gantt



7.3 Plan de acción y valoración de alternativas

El plan de acción lo podemos ver definido en el diagrama de Gantt previamente visto. En este podemos notar que existen en cada uno de los Sprint una tarea únicamente dedicada a la corrección de “errores”. Esta viene a representar los cambios que el cliente vaya demandando entre iteraciones, posibles bugs que vayamos encontrando y la mejora de la estructura del código.

Las fechas límite establecidas para cada bloque se pueden ver aplazadas por vacaciones, festivos o huelgas. En caso de que una iteración se complique o se vuelva inviable esta puede ser removida o sustituida por otra de manera relativamente barata dada la naturaleza de los proyectos “Agile” y el uso de una herramienta como GIT. Esto se debe a que cada funcionalidad de cada iteración es independiente de las otras.

Si un Sprint no avanza adecuadamente en el tiempo, todo y que contradice la propia filosofía de “Agile”, se optaría por aumentar las 20 horas semanales a 25 o 30 dependiendo del atraso acumulado.

El proyecto puede verse atrasado por la existencia de Bugs críticos que superen las 48 horas que hemos previsto para su solución. También por el deseo del cliente de cambiar algún requisito durante el desarrollo del proyecto. El tiempo sería relativo a la naturaleza de estos últimos. El impacto sería que alguna funcionalidad no crítica quedaría descartada para la versión final del proyecto y se daría prioridad a arreglar el error o la nueva funcionalidad más prioritaria. En todo caso tendríamos un software funcional (más o menos completo) por la naturaleza de trabajar usando una filosofía “Agile”.

8 Informe de sostenibilidad

Durante el transcurso del grado de ingeniería informática hemos aprendido a medir y valorar el impacto medioambiental y social en asignaturas como “Arquitectura de los computadores” y “Software Libre y Desarrollo Social”. También a gestionar proyectos en asignaturas como “Empresa y entorno económico”, “Gestión de proyectos Software” y “Ingeniería de requisitos”.

Personalmente me considero consciente de las implicaciones e impacto de los elementos mostrados en la encuesta de sostenibilidad [30], pero no tengo la experiencia fuera de lo explicado en las clases de la Universidad. Consecuentemente no conozco los métodos y/o técnicas más prácticas y adecuadas para aplicar los conocimientos planteados en la encuesta de forma adecuada sin que esto signifique una inversión de tiempo importante.

8.1 Ambiental

El impacto ambiental de este proyecto no es significativo. Esto se debe a que el desarrollo del mismo será realizado únicamente por una persona. Y para ello solo será necesario un Ordenador de escritorio, un “PC embedded” y acceso a un servidor de uso general por el CITCEA-UPC del cual el porcentaje de uso destinado al proyecto será mínimo.

Indirectamente, dada la naturaleza del proyecto, se está proporcionando una herramienta que ayude a optimizar la gestión y rendimiento energético de redes eléctricas.

8.2 Económico

Los costes del proyecto van directamente asociados al hardware y software necesario para su desarrollo además del sueldo de quien o quienes lo desarrollen. Los roles a tener en cuenta serán los de Ingeniero del Software, Ingeniero de Sistemas, “Project Manager”, Programador y “Tester”. Cualquier retraso o aumento de horas al proyecto implicará un aumento del coste de este proporcional al salario de los roles que tengan que intervenir por su causa.

8.3 Social

Personalmente, la realización de este trabajo me significa un reto y una experiencia muy importante para mi carrera como ingeniero del Software. El hecho de poder diseñar e implementar un Software de esta magnitud y el poder experimentar de primera mano las consecuencias e impacto de este sobre los proyectos del CITCEA es algo que valoró bastante positivamente.

Por otro lado, este trabajo permitirá a los ingenieros del CITCEA-UPC destinar más tiempo a innovar y realizar nuevas funcionalidades en cada proyecto. Esto implicará que se generen productos de mejor calidad para los clientes del CITCEA que buscan el rendimiento energético en sus productos. Estos podrían llegar a tener un impacto significativo en la sociedad (por ejemplo, el aumento de uso de coches eléctricos) en un caso hipotético.

9 Matriz de sostenibilidad

Los resultados mostrados a continuación se deducen de los apartados siguientes.

	PPP	Vida útil	Riesgos
Ambiental	157,2kWh	Huella ecológica	ambientales
Económico	62.920€	Plano de viabilidad	económicos
Social	Implica la adaptación a una nueva metodología de trabajo en los proyectos del CITCEA-UPC.	Impacto social	Mejora la calidad de vida del equipo técnico y de programadores

10 Dimensión económica

10.1 Costes directos

Tabla de estimación de salarios para los diferentes roles:

Rol	€/h
Software Project Manager	76 [31]
Software Engineer	90 [32]
Unix Administrator	78 [33]
Programmer	55 [34]
Tester	32 [35]

Coste total del Hardware usado en el proyecto (información proporcionada por CITCEA-UPC):

Hardware	€
HP ProDesk 600 Microtower G4 + periféricos	1.200
TS-TPC-7990	380
DiskStation DS1817+	1.400

Coste del Software usado:

Software	€
DEBIAN 9	0
Oracle Virtual Box	0
GIT	0
GitLab	0
Office 2016 Professional	0*
Windows 10	0*
Eclipse CDT	0
Libmodbus	0

*El CITCEA-UPC es un centro adscrito a la UPC por lo que el uso de las licencias de Microsoft para este proyecto es casi nulo ya que se hace uso de las licencias de la UPC.

A continuación, se mostrarán los costes directos del proyecto incluyendo únicamente aquellos que tienen un precio superior a 0€.

El cálculo de horas se deduce a partir del Gantt. Cabe destacar que cada iteración dispondrá de 4h para el "Project manager" y 4h para el "Software Engineer" debido al uso de una filosofía "Agile". El resto son para el programador.

Para calcular el coste del hardware usaremos la siguiente formula:

(Horas de uso para el proyecto) * (Precio total) / (Horas totales de vida útil)

Vida útil = 11680h para los 3 Hardware (4 años).

Costes directos	Horas	€
Project Manager	272	20.672
Software Engineer	104	9.360
Unix Administrator	72	5.616
Programmer	216	11.880
Tester	48	1.536
Total recursos humanos	712	49.064
HP ProDesk 600 Microtower G4	712	73,15
TS-TPC-7990	120	3,9
DiskStation DS1817+	20	2,4
Total Hardware	852	79,45

10.2 Costes Indirectos

Precio kwh: 0,12€ [36]

Precio de Internet: 45€/mes

Actividad	Consumo por hora
HP ProDesk 600 Microtower G4	0,1kWh
TS-TPC-7990	0,09 kWh
DiskStation DS1817+	0,2 kWh
Persona en rutina habitual	0,1 kWh

Costes indirectos	€
Electricidad	18,9
Internet	202,5
Total	221,4

10.3 Costes directos e indirectos

Tipo	€
Directo	49.143,45
Indirecto	221,4
Total	49.364,85

10.4 Costes de contingencia

Debido a que no se proveen grandes desviaciones aplicaremos un coste de contingencia del 5%.

Contingencia = 2468,24€

10.5 Costes de incidencia

Siempre existe la posibilidad de que nuestro hardware se pueda ver afectado por alguna incidencia y/o que las previsiones de tiempo para realizar las tareas sean más costosas de programar que las previstas inicialmente.

Los costes serían un atraso del proyecto y el pagar un día entero de trabajo al administrador de sistemas o al programador dependiendo del tipo de incidencia.

Causa	Solución	Riesgo (%)	Coste de impacto (€)	Coste (€)
Daño en el Hardware	Sustitución de la pieza y restaurar el entorno de trabajo	1%	624+(coste de la pieza + envío)	80
Mala previsión	Añadir horas de programación	20%	440	88
Total		0,2%	1064	168

10.6 Coste Total

El coste total del proyecto quedará de la siguiente manera:

Tipo	Coste (€)
Directo e Indirecto	49.364,85
Contingencia	2468,24
Incidencia	168
Total	52.000
Total + Impuestos (21%)	62.920

10.8 Control y desviación

Los desvíos que debemos considerar en nuestro proyecto se resumen en una variación en las horas reales del proyecto (hecho que aumentaría o disminuiría proporcionalmente el coste), en posibles reemplazos del hardware del proyecto y sobre el coste real de la mano de obra (la estimación no corresponde con el sueldo real).

10.9 Conclusión

El proyecto puede parecer costoso, pero si consideramos que durante su vida útil el coste que reducirá de los futuros proyectos del CITCEA-UPC es bastante viable. Además, la arquitectura del software está pensada para poder incluir fácilmente en futuras versiones código externo en caso de que sea necesario. En resumen, la mayor parte del coste se debe a la naturaleza de ser un proyecto piloto y tener que integrarse y adaptarse.

11 Anexo

11.1 Patrón de diseño

Un patrón de diseño sistemáticamente nombra, motiva y explica un diseño de propósito general que se asocia a un problema de diseño recurrente en sistemas orientados a objetos. Este describe el problema, la solución, cuando aplicarlo y las consecuencias de su uso.

11.2 Software

El Software se define como un conjunto de instrucciones computacionales que pueden ser ejecutadas en un orden deseado con la finalidad de obtener un funcionalidad y rendimiento específico de un computador. También como las estructuras que permiten al programa manipular adecuadamente los datos y la documentación que describe las operaciones y uso del mismo.

11.3 PC Embedded

Particularmente para este proyecto, cuando se haga mención de un PCE, estaremos hablando sobre un computador que estará empotrado junto a los equipos de electrónica de potencia y conectado a dispositivos de bajo nivel.

11.4 Dispositivo de bajo nivel

Dispositivos que nos permiten consultar configurar y/o consultar información sobre elementos físicos de los equipos de electrónica de potencia. Por la naturaleza de este proyecto nos abstraeremos sobre los detalles relacionados sobre estos y nos centraremos únicamente como comunicarnos con ellos apropiadamente.

11.5 Interface

Interface se define como un límite común entre dos o más componentes de sistemas computacionales que intercambian información (Software, periféricos, humanos, etc).

12 Referencias

[1] MOHAN, N. (2003). First Course on Power Electronics and Drives.

[2] CITCEA. [En línea]. [Consultado: 31 Marzo 2019]. Disponible en Internet: <https://www.citcea.upc.edu/>

[3] Prácticas en empreses | Facultad de informàtica de Barcelona. [En línea]. [Consultado el 31 Marzo 2019]. Disponible en Internet: <https://www.fib.upc.edu/ca/empresa/practiques-en-empresa>

[4] Corrigan, S. (2002). Introduction to the Controller Area Network (CAN). [en línea]. [Consultado: 19 Marzo 2019]. Disponible en Internet: <http://www.ti.com/lit/an/sloa101b/sloa101b.pdf >.

[5] The Modbus Organization. [en línea]. [Consultado: 19 Marzo 2019]. Disponible en Internet: <http://www.modbus.org>.

[6] The GNU General Public License v3.0. [en línea]. [Consultado: 17 Junio 2019]. Disponible en Internet: <https://www.gnu.org/licenses/gpl-3.0.en.html>.

[7] ADFWeb.com Gateway CANopen, Modbus, TCP. [En línea]. [Consultado: 31 Marzo 2019]. Disponible en Internet: <https://www.adfweb.com/home/default.asp>

[8] Anybus Gateway Index Page. [En línea]. [Consultado: 31 Marzo 2019]. Disponible en Internet: <https://www.anybus.com/products/gateway-index>

[9] Libmodbus.org - News. [en línea]. [Consultado: 19 Marzo 2019]. Disponible en Internet: <http://www.libmodbus.org>

[10] SocketCAN. [En línea]. [Consultado: 31 Marzo 2019]. Disponible en Internet: <https://www.kernel.org/doc/Documentation/networking/can.txt>

[11] Qt CAN Bus | Qt Serial Bus 5.12.3. [En línea]. [Consultado: 31 Marzo 2019]. Disponible en Internet: <https://doc.qt.io/qt-5/qtcanbus-backends.html>

[12] Python-can · PyPI. [En línea]. [Consultado: 31 Marzo 2019]. Disponible en Internet: < https://pypi.org/project/python-can/>

[13] QModMaster. [En línea]. [Consultado: 31 Marzo 2019]. Disponible en Internet: <https://bitbucket.org/codeimproved/qslog/src/master/>.

[14] Larman, C, Prentice Hall (2005). Applying UML and patterns: *an introduction to object-oriented analysis and design and iterative development*.

[15] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J, Addison-Wesley (1995). Design patterns: elements of reusable object-oriented software.

[16] Beck, K. (2000). Extreme Programming Explained: Embrace Change.

[17] Booch, G; Rumbaugh, J; Jacobson, I. (2005). Unified Modeling Language User Guide.

[18] Corey, L. (2009). Scrumban: Essays on Kanban Systems for Lean Software Development. Modus Cooperandi Press.

[19] Pressman, R.S.; Maxim, B.R (2015). Software engineering: a practitioner's approach. McGraw Hill Higher Education.

[20] Martin Fowler (1999). Refactoring: Improving the Design of Existing Code. [En línea]. [Consultado: 19 Marzo 2019] <https://www.refactoring.com>.

[21] Scott Chacon, Ben Straub (2014). Pro Git. Everything you need to know about GIT.

[22] Debian -- About. [En línea]. [Consultado: 20 Marzo 2019]. <https://www.debian.org/intro/about>.

[23] ECLIPSE CDT [En línea]. [Consultado: 20 Marzo 2019]. <https://www.eclipse.org/cdt/>.

[24] Autotools Introduction (automake) [En línea]. [Consultado: 20 Marzo 2019]. <https://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html>.

[25] TS-TPC-7990 – Technologic Systems Manuals [En línea]. [Consultado: 20 Marzo 2019]. <https://wiki.embeddedarm.com/wiki/TS-TPC-7990>.

[26] Internet Assigned Number Authority [En línea]. [Consultado: 20 Marzo 2019]. <https://www.iana.org/>.

[27] Modbus Communication – MATLAB & Simulink [En línea]. [Consultado: 20 Marzo 2019]. <https://www.mathworks.com/help/instrument/modbus-communication.html>.

[28] GitLab [En línea]. [Consultado: 20 Marzo 2019]. < https://about.gitlab.com/>.

[29] TS-TPC-7990 Embedded Touch Panel Powered by ARM [En línea]. [Consultado: 20 Marzo 2019]. < https://www.embeddedarm.com/products/TS-TPC-7990>.

[30] Cuestionario de Estudiantes de Ingeniería Informática. [En línea]. [Consultado: 20 Marzo 2019]. < goo.gl/kWLMLE>.

[31] Salary for Software Product Manager. [En línea]. [Consultado: 20 Marzo 2019]. <https://www1.salary.com/software-product-manager-i-salary.html>.

[32] Salary for Software Engineer III. [En línea]. [Consultado: 20 Marzo 2019]. < https://swz.salary.com/SalaryWizard/Software-Engineer-III-Salary-Details.aspx>.

[33] Salary for Unix Systems Administrator. [En línea]. [Consultado: 20 Marzo 2019]. < https://swz.salary.com/SalaryWizard/UNIX-Systems-Administrator-Salary-Details.aspx>.

[34] Salary for Application Programmer I. [En línea]. [Consultado: 20 Marzo 2019]. < https://swz.salary.com/salarywizard/Applications-Programmer-I-Salary-Details.aspx>.

[35] Salary for Tester I. [En línea]. [Consultado: 20 Marzo 2019]. <https://swz.salary.com/SalaryWizard/Software-Quality-Assurance-Tester-I-Salary-Details.aspx >.

[36] PVPC | ESIOS electricidad · datos · transparencia. [En línea]. [Consultado: 20 Marzo 2019]. <https://www.esios.ree.es/es/pvpc>.